ENTHOUGHT
SCIENTIFIC COMPUTING SOLUTIONS

The Enthought Developer Tool Suite

David C. Morrill, Enthought, Inc.

Version 1, 22-Dec-06

Enthought, Inc.
515 Congress Avenue
Suite 2100
Austin TX 78701
1.512.536.1057 (voice)
1.512.536.1059 (fax)
http://www.enthought.com
info@enthought.com

Table of Contents

1 Introduction

The *Enthought Developer Tool Suite* is a collection of Envisage plug-ins intended to make the development and debugging of Envisage-based applications easier for developers.

The plug-ins heavily use the **DockWindow** *features* capability, and in fact were originally developed as a test bed for exploring the design space that the feature API provides. However, the resulting collection of finished plug-ins proved so useful that they are being released as an independent package themselves.

Note that the *Enthought Developer Tool Suite* should not be confused with the *Enthought Tool Suite*, which is the general term used to describe the entire code base that spans the spectrum of Enthought's open source tools and packages. The *Enthought Developer Tool Suite* focuses solely on developer-oriented tools for use within Envisage Workbench-based applications.

A secondary goal of the tool suite is to also provide interesting examples of how to design and use features based on the **DockWindowFeature** API. Curious developers are encouraged to study the source of the tools in the package (**enthought.developer.tools**), and to learn more about the *feature* architecture in *The DockWindowFeature DockWindow Feature* documentation and source (**enthought.pyface.dock.features**).

1.1 Adding the Tool Suite to an Envisage Application

The tool suite includes two plug-in definition files that can be included into the plug-in definitions file for your Envisage application. A partial listing of a sample Envisage plug-ins definition file which includes both plug-in definition files is shown below:

```
from os.path import abspath, dirname, join

# Package location:
enthought = abspath( dirname( enthought.__file__ ) )

# The plugin definitions required by the application:
PLUGIN_DEFINITIONS = [
    # Your application plugins:
    …

    # Enthought developer tool suite plugins:
    join( enthought, 'developer/plugin_definition.py' ),

    # Enthought FBI debugger plugin:
    join( enthought, 'developer/fbi_plugin_definition.py'
),
]

#  Plug-in definitions that we want to import from but
don't want
#  as part of the application:
INCLUDE = []
```

Including the `plugin_definition.py` file adds the following items to your application:

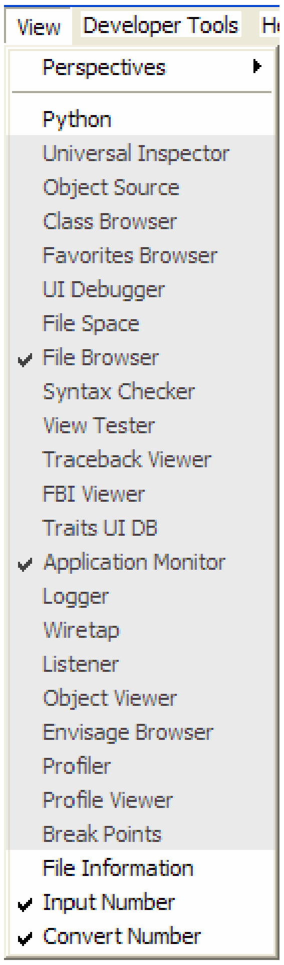
- All of the developer tool suite plug-ins.
- All of the DockWindow *features* used by the plug-ins.
- A new Developer Tools perspective.
- A **Developer Tools** menu on the main application men bar.

Including the `fbi_plugin_definition.py` file adds a hook that allows the FBI (Frame Based Inspector) debugger to handle unhandled exceptions that occur in your application.

If for some reason you do not wish to include all of the plug-ins, simply copy and paste the ones you wish to use into a new plug-in definition file and include that file instead.

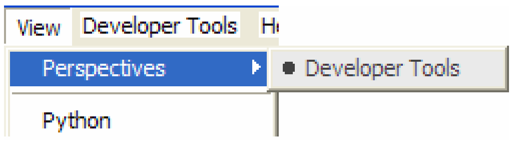
1.2 Using the Plug-ins

Once the plug-ins have been added to your Envisage application, they appear on the **View** menu of your main application tool bar, as shown in the following figure:



In this figure, the developer tool suite plug-ins are shown shaded in grey. Selecting or unselecting a plug-in on the menu adds or removes the tool from the current perspective. You can then reposition or resize the tool using the standard **DockWindow** splitter bars, drag bars and notebook tabs.

If you like, you can also switch to the supplied Developer Tools perspective by selecting it from the **View > Perspectives** menu, as shown below.



2 The Developer Tool Suite Plug-ins

The following sections describe the function and use of each of the developer tool plug-ins. The information includes:

- What the tool does and how to use it.
- Screen shots of the tool in use.
- Tips on how to use the plug-in in combination with other plug-ins.

The last item is especially important, because very few, if any, of the plug-ins are designed to be used in isolation. Each tool typically performs one or two functions, but is written so that it can be easily connected to other tools. In many ways, the underlying model is much like the UNIX philosophy of "small, sharp tools", which have well-defined functions and support myriad ways of recombining them to create new, more powerful tools.

Here is a list of the tools and a synopsis of what each one does:

- **Application Monitor**: Displays top-level Envisage application objects.
- **Break Points**: Manages all currently-set debugger breakpoints.
- **Class Browser**: Displays module, class, and method information.
- **Envisage Browser**: Displays Envisage plug-in information.
- **Favorites Browser**: Displays names of commonly used files.
- **FBI Viewer**: Displays Python source code and allows setting breakpoints.
- **File Browser**: Displays a file system hierarchy.
- **File Space**: Displays a union of disjoint file system hierarchies.
- **Listener**: Displays all listeners for a specified trait.
- **Logger**: Displays and filters log messages.
- **Object Source**: Displays the source code for a specified object.
- **Object Viewer**: Displays the contents of Python objects.
- **Profiler**: Profiles the execution of specified methods.
- **Profile Viewer**: Displays the results of profiled execution.
- **Syntax Checker**: Checks a Python source file for syntax errors.
- **Traceback Viewer**: Displays an exception traceback.

- **Traits UI DB**: Displays the contents of the Traits UI data base.
- **UI Debugger**: Displays the window object hierarchy for a specified window.
- **Universal Inspector**: Displays Python object data or file contents.
- **View Tester**: Testing harness for Traits UI views.
- **Wiretap**: Sets breakpoints on trait changes.

The following table lists the types of objects that the various tools operate on. "Drag Source" and "Connect From" indicate the types of data that a tool can send to another tool; "Drop Target" and "Connect To" indicate the types of data that a tool can receive from another tool.

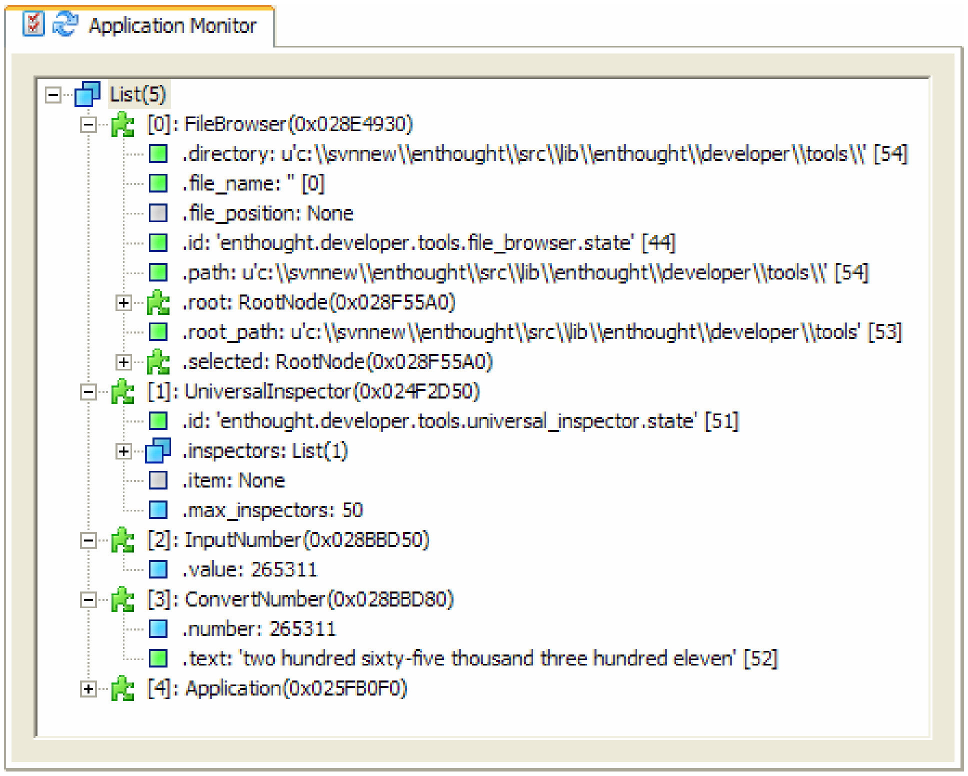
Table 1 Types of data displayed and shared by Enthought Developer Tools

Tool	*Displays*	*Drag Source*	*Drop Target*	*Connect From*	*Connect To*
Application Monitor	Python objects	Python object			
Break Points	Breakpoints	File position		File position	
Class Browser	Packages, modules, classes, methods	File position		File position	
Envisage Browser				Application object, extension point, query result	
Favorites Browser	Modules, classes, methods	File position	Python file	File position	
FBI Viewer	Source code	File name			File position
File Browser	Directories, files	File position, file, path, directory		File position, file, path, directory	
File Space	Directories, Python files	File position, file, path, directory		File position, file, path, directory	

Tool	Displays	Drag Source	Drop Target	Connect From	Connect To
Listener	Listener objects	File position, selected object	Trait	File position	Trait
Logger	Log messages	File position		File position, traceback text	
Object Source	Modules, classes, methods	File position	Traited object	Traited object, file position	Traited object
Object Viewer	Traited objects		Traited object		Traited object
Profiler	Methods	Results file name	Class, method	Results file name	Class, method
Profile Viewer	Profiler results	Results file position		Result entry	Profiler results file
Syntax Checker	Python source code	Python file	Python file		Python file
Traceback Viewer	Tracebacks	Traceback file position	Traceback	File position	Traceback text
Traits UI DB	Traits UI database items	Selected value		Selected value	
UI Debugger		Selected object	Window object	Selected value	
Universal Inspector	Python objects, pickled objects, binary files, text files	Object, file name	Python object or value	Python object or value	Python object or value
View Tester	Traits UI views		Python file		
Wiretap	Wiretap breakpoints		Trait		

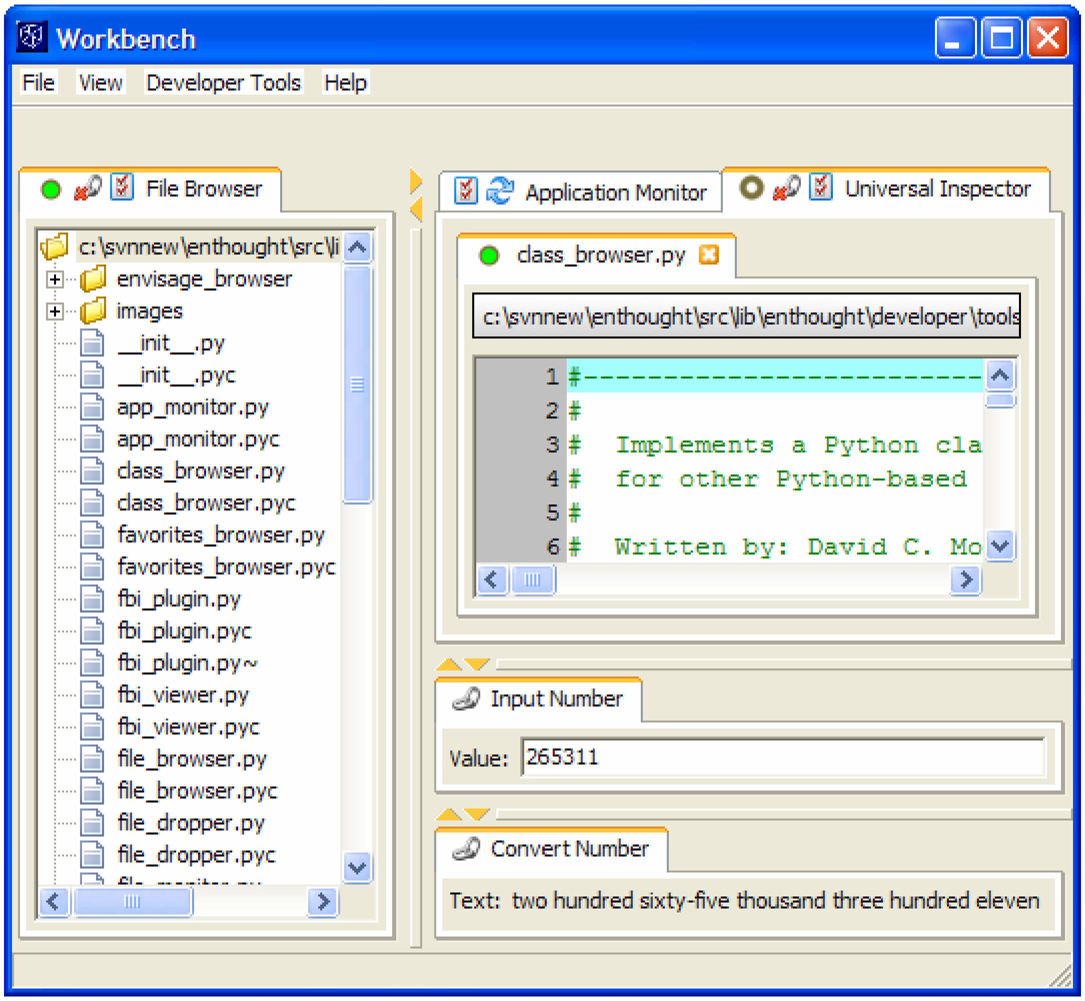
2.1 Application Monitor

The Application Monitor displays the contents of each Python object bound to a **View** or **Editor** in the containing Envisage **Workbench** window. In practice, this typically means that it displays the contents of all objects used to create an Envisage view using a UOL (Universal Object Locator) value. Here is a screen shot of a typical Application Monitor view:



```
List(5)
    [0]: FileBrowser(0x028E4930)
        .directory: u'c:\\svnnew\\enthought\\src\\lib\\enthought\\developer\\tools\\' [54]
        .file_name: '' [0]
        .file_position: None
        .id: 'enthought.developer.tools.file_browser.state' [44]
        .path: u'c:\\svnnew\\enthought\\src\\lib\\enthought\\developer\\tools\\' [54]
        .root: RootNode(0x028F55A0)
        .root_path: u'c:\\svnnew\\enthought\\src\\lib\\enthought\\developer\\tools' [53]
        .selected: RootNode(0x028F55A0)
    [1]: UniversalInspector(0x024F2D50)
        .id: 'enthought.developer.tools.universal_inspector.state' [51]
        .inspectors: List(1)
        .item: None
        .max_inspectors: 50
    [2]: InputNumber(0x028BBD50)
        .value: 265311
    [3]: ConvertNumber(0x028BBD80)
        .number: 265311
        .text: 'two hundred sixty-five thousand three hundred eleven' [52]
    [4]: Application(0x025FB0F0)
```

The following screen shot shows the Envisage workbench application window that the Application Monitor in the previous screen shot is monitoring. Each view (top-level tab) in the Workbench window is represented by a top-level node in the Application Monitor.



c:\svnnew\enthought\src\li
envisage_browser
images
__init__.py
__init__.pyc
app_monitor.py
app_monitor.pyc
class_browser.py
class_browser.pyc
favorites_browser.py
favorites_browser.pyc
fbi_plugin.py
fbi_plugin.pyc
fbi_plugin.py~
fbi_viewer.py
fbi_viewer.pyc
file_browser.py
file_browser.pyc
file_dropper.py
file_dropper.pyc

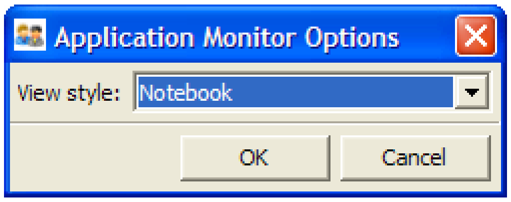
c:\svnnew\enthought\src\lib\enthought\developer\tools

```
#------------------------
#
#  Implements a Python cla
#  for other Python-based
#
#  Written by: David C. Mo
```

Value: 265311

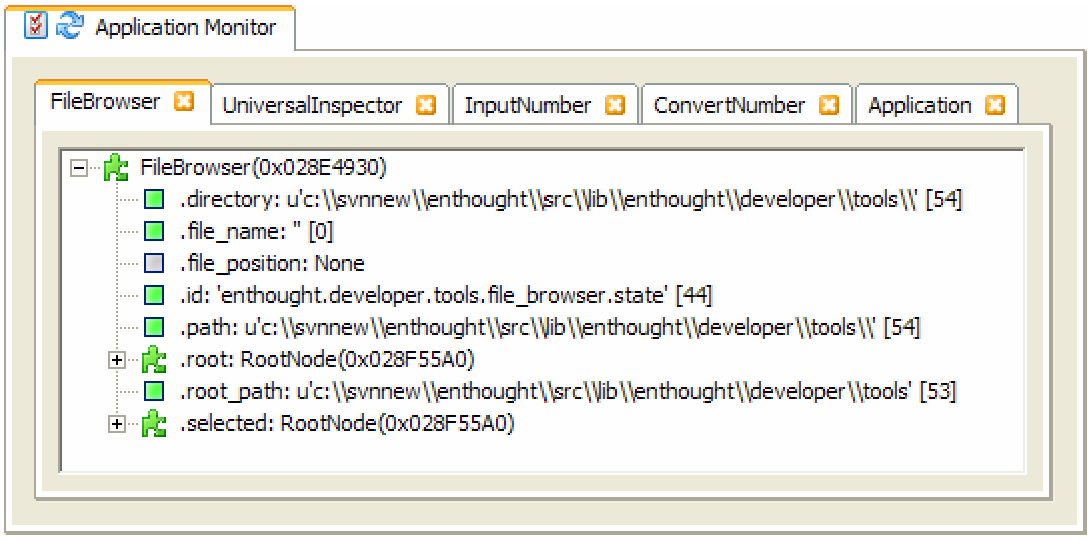
Text: two hundred sixty-five thousand three hundred eleven

Clicking the options icon (☑) on the **Application Monitor** tab displays the following dialog box:



View style: Notebook

This dialog box allows you to choose between showing all of the objects in a single tree view as individual elements within a top-level list (as shown in the first screen shot), or as separate pages in a notebook, as shown in the following screen shot:



```
FileBrowser(0x028E4930)
  .directory: u'c:\\svnnew\\enthought\\src\\lib\\enthought\\developer\\tools\\' [54]
  .file_name: '' [0]
  .file_position: None
  .id: 'enthought.developer.tools.file_browser.state' [44]
  .path: u'c:\\svnnew\\enthought\\src\\lib\\enthought\\developer\\tools\\' [54]
  .root: RootNode(0x028F55A0)
  .root_path: u'c:\\svnnew\\enthought\\src\\lib\\enthought\\developer\\tools' [53]
  .selected: RootNode(0x028F55A0)
```

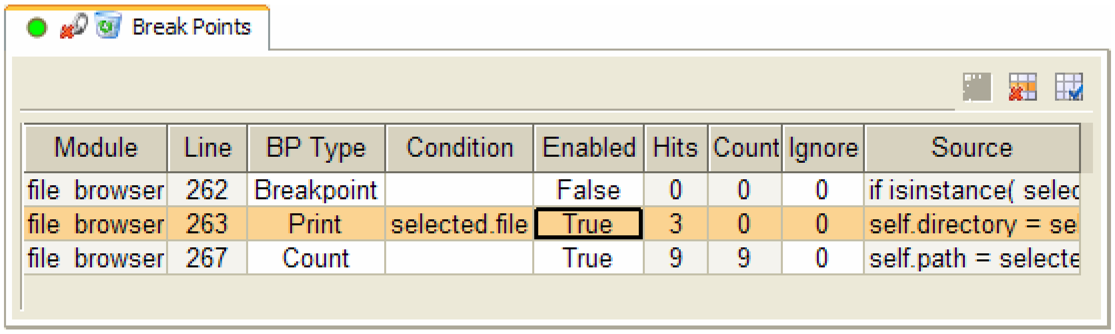
When in notebook mode, you can delete any object from the view by clicking its tab's close icon (☒).

You can update the contents of the tool when new views are added to the workbench window by clicking the refresh icon (🔄) in the **Application Monitor** tab.

All tree-view items within the tool are draggable. For example, you can drag an object contained in the view to the Object Source tool in order to view the source code of the object's class.

2.2 Break Points

The Break Points tool allows you to manage all of the breakpoints currently set in your code. Here is a screen shot of a typical Break Points view:



Module	Line	BP Type	Condition	Enabled	Hits	Count	Ignore	Source
file_browser	262	Breakpoint		False	0	0	0	if isinstance(selec
file_browser	263	Print	selected.file	True	3	0	0	self.directory = se
file_browser	267	Count		True	9	9	0	self.path = selecte

The view shows a table of all current breakpoints, which contains the following columns:

- **Module**: The name of the module that contains the breakpoint.
- **Line**: The number of the line of the module that the breakpoint is set on.
- **BP Type**: The type of the breakpoint, which can be one of the following values:
    - **Breakpoint**: Stops execution each time the breakpoint is hit.
    - **Temporary**: Stops execution the next time the breakpoint is hit, and then removes the breakpoint.
    - **Count**: Counts the number of times the breakpoint is hit, but does not stop execution.
    - **Trace**: Executes the contents of the **Condition** field each time the breakpoint is hit, but does not stop execution.
    - **Print**: Prints the current value of the **Condition** field each time the breakpoint is hit, but does not stop execution.
    - **Log**: Logs the current value of the **Condition** field each time the breakpoint is hit, but does not stop execution.
- **Condition**: An optional Python expression or statement that affects the handling of the breakpoint. The handling of the condition depends upon the value of **BP Type** as follows:
    - **Breakpoint**, **Temporary**, **Count**: An expression that must evaluate to **True** in order for the breakpoint to be hit or counted.
    - **Trace**: A Python statement that is executed each time the breakpoint is hit.
    - **Print**, **Log**: An expression that is evaluated and printed or logged each time the breakpoint is hit.

- **Enabled**: If **True**, the breakpoint is enabled; otherwise it is disabled and has no effect.
- **Hits**: The number of times the breakpoint has been hit.
- **Count**: The number of times the breakpoint has been hit (only changes when the **BP Type** is **Count**).
- **Ignore**: The number of times the breakpoint will be ignored before it is triggered.
- **Source**: The content of the line that the breakpoint is set on.

You can change the **BP Type**, **Condition**, **Enabled** and **Ignore** fields in the tool by clicking the field and entering a new value.

To delete an individual breakpoint, select its row in the table and click the **Delete row** icon () in the table toolbar. To delete all breakpoints at once, click the dispose icon () on the **Break Points** tab.

Changes to the breakpoints are persisted immediately. If you run the Envisage application at a later time, you will still have access to all of the breakpoints that you previously set. However, the breakpoints are not automatically loaded when the application is started. This behavior allows you to run the application normally, without constantly popping into the FBI debugger. To reload any previously set breakpoints, do one of the following:

- Click the reload icon () on the **Break Points** tab. If the icon is not present, the breakpoints have already been loaded.
- On the **Developer Tools > Debug** menu, select **Restore break points**.

If you wish, you can sort the breakpoints in various ways by clicking the column header of the field you wish to sort on. To sort in the opposite order, click the same column header again. You remove sorting by clicking the **Unsort** icon () in the table toolbar.

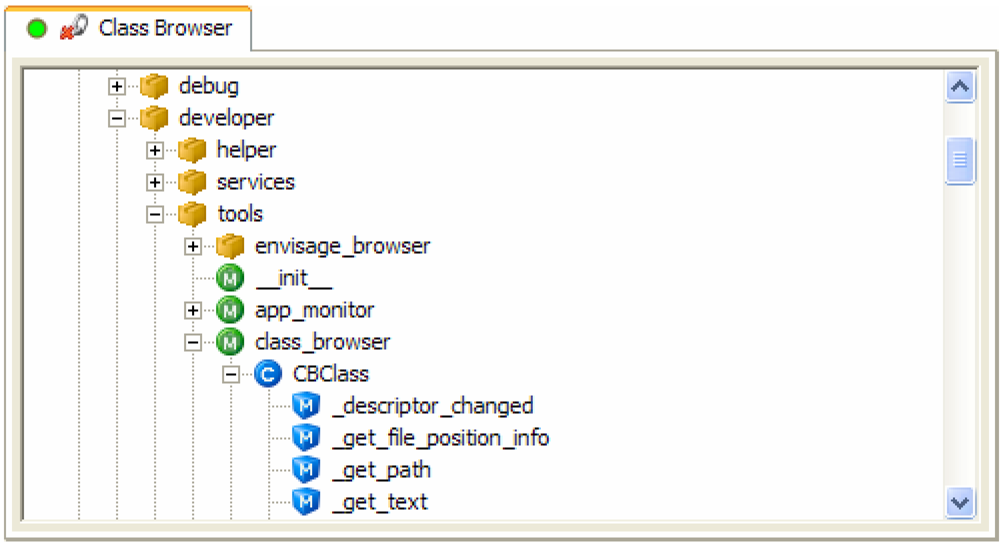
You can reorganize, delete, and add table columns by clicking the user preference icon () in the table toolbar, and modifying the structure of the table using the dialog box that appears.

Each time you select a breakpoint in the table, you also select its corresponding source code file position. You can provide this information to other tools by dragging the drag icon (●) and dropping it on another tool, or by connecting the current source code file position to another tool by dragging or clicking the connection icon (). Both icons are located on the **Break Points** tab.

For example, if you forget the details of where particular breakpoints are set, you might want to connect the Break Points tool to the FBI Viewer so that you can view the corresponding source code by clicking breakpoints in the Break Point tool.

2.3 Class Browser

The Class Browser tool displays a tree view of the source code in your Python path; each root in the Python path expands to the module, class, and method levels. A screen shot of the Class Browser in use is shown below:



Each time you start the Class Browser tool, it scans your Python path, analyzes all of the Python modules it finds, and stores the results in a database. Starting for the first time in a session takes the longest time, because the database does not exist and must be built from scratch. Thereafter, the process is usually much faster, because the tool needs to analyze only new or changed source files. The scanning process is also performed periodically as long as the Class Browser view is open.

Items in the Class Browser, such as modules, classes, and methods can also be supplied to other tools using one of the following techniques:

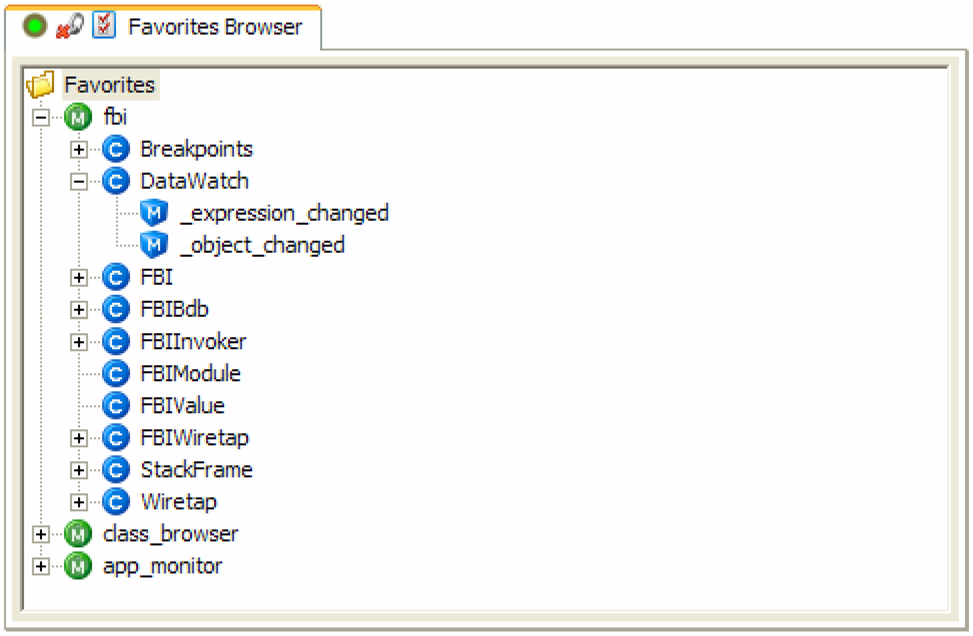
- Drag an item in the tree view and drop it on another tool, such as the Universal Inspector.
- Select an item in the Class Browser tree view then drag the browser's drag, ●, icon and drop it on another tool.
- Connect the Class Browser to another tool by dragging or clicking the connect icon () in the **Class Browser** tab. After that, when you select items in the Class Browser tree view, the Class Browser automatically sends the selected item to the connected tool.

2.4 Envisage Browser

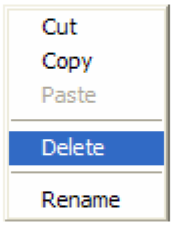
TO BE WRITTEN…

2.5 Favorites Browser

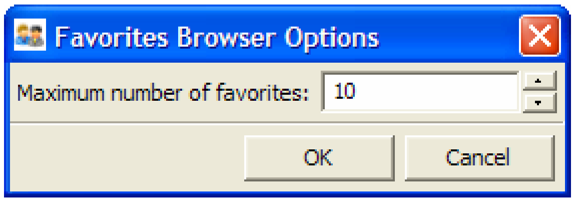
The Favorites Browser tool displays a list of your "favorite" Python modules using a class browser style tree view of modules, classes, and methods. Developers often work on a small subset of files, perhaps stored across several different packages. The Favorites Browser allows this current working set of files to be organized into a single view. The following screen shot shows an example of the Favorites Browser in use:



To add Python modules the Favorites Browser, drag them from any file source and drop them on the drag and drop icon (●) in the tool’s tab. To delete modules from the view, right-click the module in the browser view and click **Delete** on the shortcut menu:



By default, the Favorites Browser displays the ten most recently added Python modules. To change the number of items, click the options icon (☑) in the browser tab, which displays the following dialog box:



Maximum number of favorites: 10

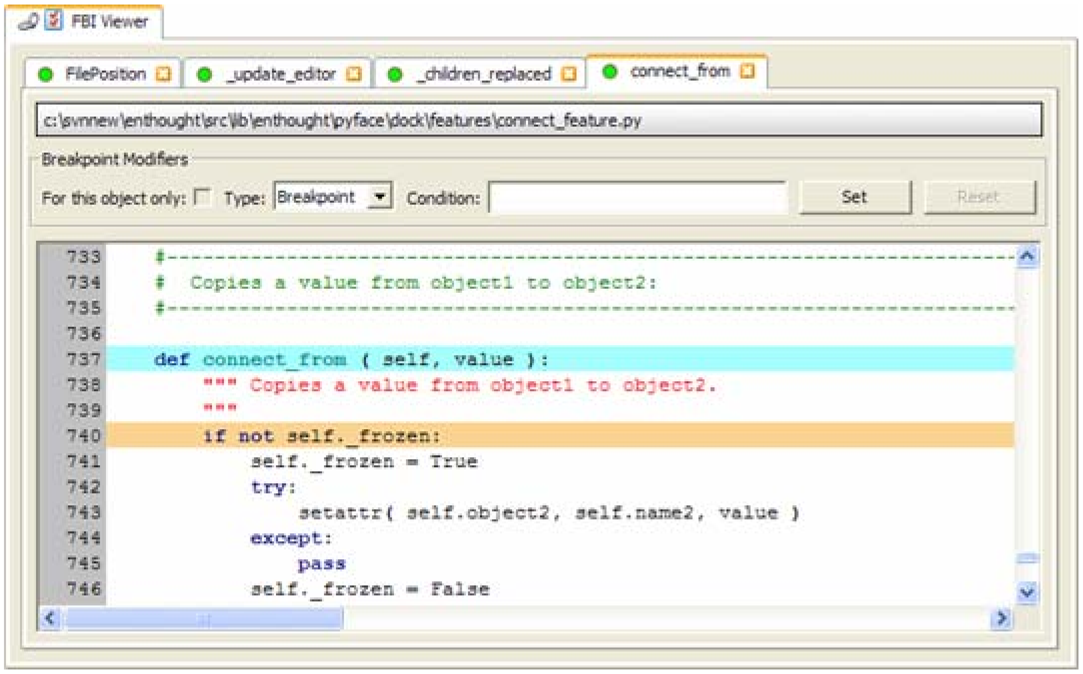
Change **Maximum number of favorites** to a new value in the range from 1 to 50, and then click **OK**.

Selecting any item in the browser's tree view also selects a **FilePosition** object that describes the section of Python code that corresponds to the item (i.e., module, class or method). You can make this **FilePosition** available to other tools by doing one of the following:

- Drag the browser's drag and drop icon (●) and drop it on another tool that accepts **FilePosition** information, such as the FBI Viewer.
- Drag or click the browser's connect icon ( ) and connect the browser to another tool that accepts **FilePosition** information.

2.6 FBI Viewer

The FBI Viewer (Frame Based Inspector Viewer) displays Python source code and allows setting and removing breakpoints within the code. Here is a screen shot of the FBI Viewer in use:



c:\svnnew\enthought\src\lib\enthought\pyface\dock\features\connect_feature.py

```
    #-------------------------------------------------------------------------------
    #  Copies a value from object1 to object2:
    #-------------------------------------------------------------------------------

    def connect_from ( self, value ):
        """ Copies a value from object1 to object2.
        """
        if not self._frozen:
            self._frozen = True
            try:
                setattr( self.object2, self.name2, value )
            except:
                pass
            self._frozen = False
```

You can specify the Python source code to display by connecting the FBI Viewer to a tool that supplies source files, such as the File Browser, Class Browser, Object Source or Listener tools.

Each new source file or file fragment that you add to the viewer creates a new sub-tab for displaying the source code. Each sub-tab contains:

- A message bar at the top displaying the fully qualified name of the source file.
- A set of controls for adding and removing breakpoints.
- A read-only text area for viewing the source code and selecting breakpoints.

You can close any sub-tab by clicking its close icon (⊠), and you can drag the source code object to another tool by dragging the sub-tab's drag icon (●).

To set breakpoints, select the appropriate source code line, and then click **Set**. To remove a breakpoint, select a source code line that has a breakpoint in place, and click **Reset**.

The FBI Viewer supports several different types of breakpoints:

- **Breakpoint**: Stops execution when the breakpoint is hit.
- **Temporary**: Stops execution when the breakpoint is hit, then removes the breakpoint.
- **Count**: Count the number of times that the breakpoint is hit, but does not stop execution.
- **Trace**: Executes the Python statement specified by **Condition** each time the breakpoint is hit, but does not stop execution.
- **Print**: Prints the value of the Python expression specified by **Condition** each time the breakpoint is hit, but does not stop execution.
- **Log**: Logs the value of the Python expression specified by **Condition** each time the breakpoint is hit, but does not stop execution.

To specify the breakpoint type, select it in the **Type** list.

The contents of the **Condition** field are used differently, depending upon the type of breakpoint set:

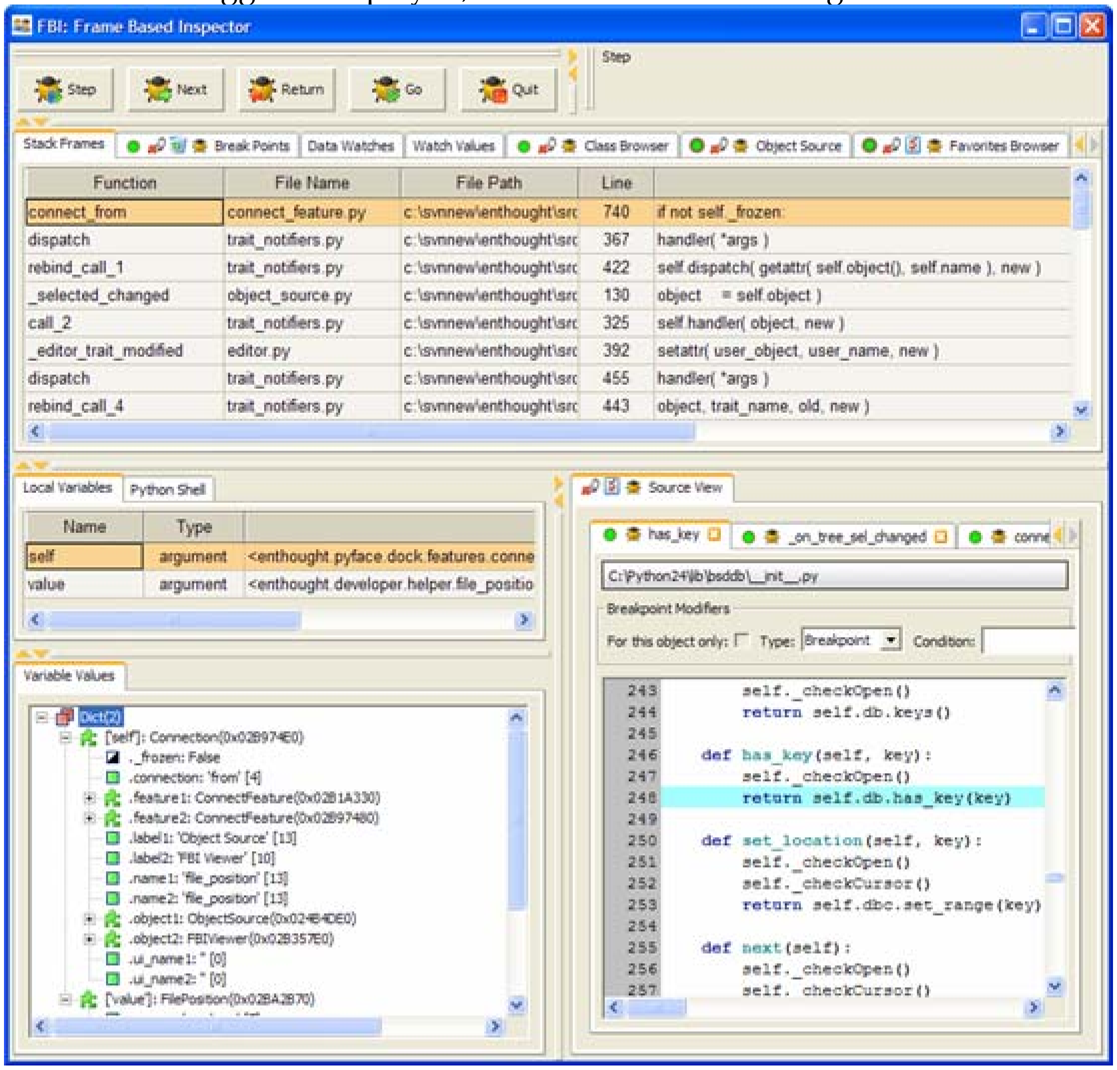
- **Breakpoint**, **Temporary**, **Count**: An expression that must evaluate to **True** in order for the breakpoint to be hit or counted. No value specified means the breakpoint is always hit.

- **Trace**: A Python statement that is executed each time the breakpoint is hit.

- **Print**, **Log**: A Python expression that is evaluated and printed or logged each time the breakpoint is hit.

You can select **For this object only** if you want the breakpoint to be triggered only for the specified object. This option is available only when the source code supplied to the FBI Viewer has an associated object. For example, the Listener tool creates **FilePosition** objects with the listener object attached to it.

When a **Breakpoint** or **Temporary** breakpoint is triggered, the FBI debugger is displayed, as shown in the following screen shot:



Function	File Name	File Path	Line	
connect_from	connect_feature.py	c:\svnnew\enthought\src	740	if not self._frozen:
dispatch	trait_notifiers.py	c:\svnnew\enthought\src	367	handler(*args)
rebind_call_1	trait_notifiers.py	c:\svnnew\enthought\src	422	self.dispatch(getattr(self.object(), self.name), new)
_selected_changed	object_source.py	c:\svnnew\enthought\src	130	object = self.object)
call_2	trait_notifiers.py	c:\svnnew\enthought\src	325	self.handler(object, new)
_editor_trait_modified	editor.py	c:\svnnew\enthought\src	392	setattr(user_object, user_name, new)
dispatch	trait_notifiers.py	c:\svnnew\enthought\src	455	handler(*args)
rebind_call_4	trait_notifiers.py	c:\svnnew\enthought\src	443	object, trait_name, old, new)

Name	Type	
self	argument	<enthought.pyface.dock.features.conne
value	argument	<enthought.developer.helper.file_positio

```
Dict(2)
  ['self']: Connection(0x02B974E0)
    ._frozen: False
    .connection: 'from' [4]
    .feature1: ConnectFeature(0x02B1A330)
    .feature2: ConnectFeature(0x02B97480)
    .label1: 'Object Source' [13]
    .label2: 'FBI Viewer' [10]
    .name1: 'file_position' [13]
    .name2: 'file_position' [13]
    .object1: ObjectSource(0x024B4DE0)
    .object2: FBIViewer(0x02B357E0)
    .ui_name1: '' [0]
    .ui_name2: '' [0]
  ['value']: FilePosition(0x02BA2B70)
```

C:\Python24\lib\bsddb__init__.py

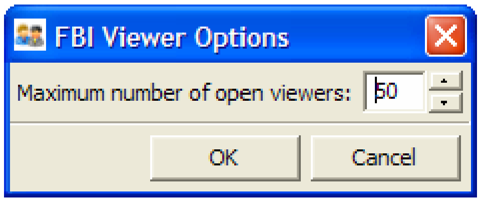
For this object only: ☐ Type: Breakpoint Condition:

```
        self._checkOpen()
        return self.db.keys()

    def has_key(self, key):
        self._checkOpen()
        return self.db.has_key(key)

    def set_location(self, key):
        self._checkOpen()
        self._checkCursor()
        return self.dbc.set_range(key)

    def next(self):
        self._checkOpen()
        self._checkCursor()
```

Refer to the FBI debugger documentation for more information on how to use it.

You can view information generated by **Count** breakpoints using the **Break Points** tool, log messages generated using **Log** breakpoints using the Logger tool, and information generated by **Trace** and **Print** breakpoints by using the standard Envisage Python shell.

By default, the FBI Viewer shows a maximum of 50 source code tabs at one time. You can change this value by clicking the viewer's options icon (), which displays the following dialog box:

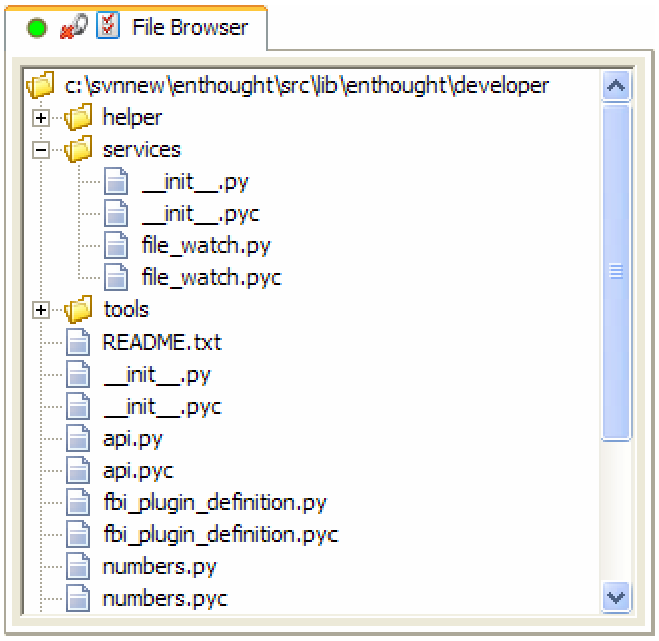


Maximum number of open viewers: 50

Change **Maximum number of open viewers** to whatever value you like, and then click **OK**. When you open a new tab that would exceed the maximum value, the oldest tab is automatically closed. If you specify a maximum that is smaller than the current number of open tabs, the tool closes the required number of oldest tabs.

2.7 File Browser

The File Browser tool shows a hierarchical tree view of all or part of the file system. Here is screen shot of the File Browser in use:



The main purpose of the File Browser is to act as a source of file names and information for other tools, so it does not support any actions, such as delete or rename, on the files or directories it displays.

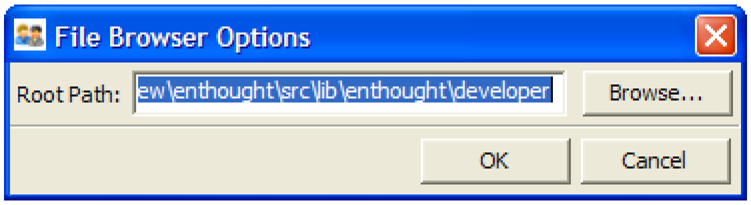
Selecting a file or directory in the File Browser makes the following information available to other tools:

- File name (when a file is selected)
- **FilePosition** object (when a file is selected)
- Directory name (when a directory is selected)
- File object (when either a directory or file is selected)

You can share this information about the current selection with other tools by doing one of the following:

- Drag the File Browser's drag icon (●) and drop it on another tool.
  Connect the File Browser to another tool by dragging or clicking its connect icon ( ).

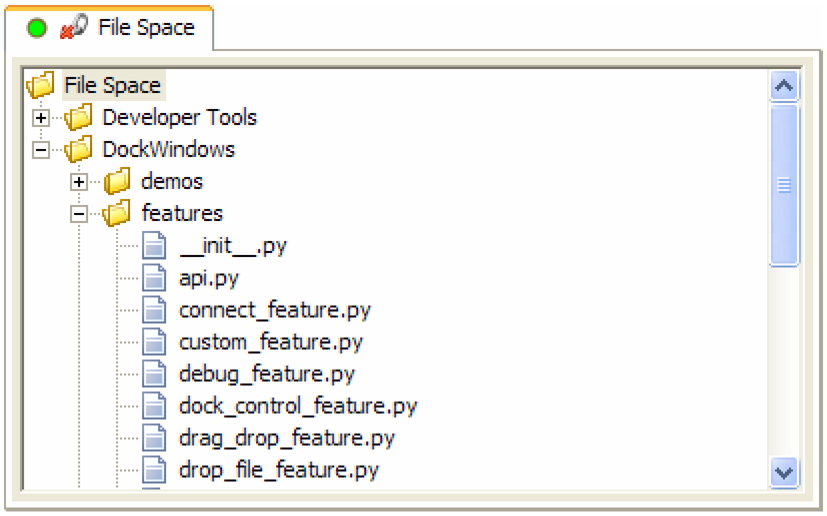
Any directory within the file system can serve as the root of the File Browser view. To change the current root directory, click the browser's options icon ( ), which displays the following dialog box:



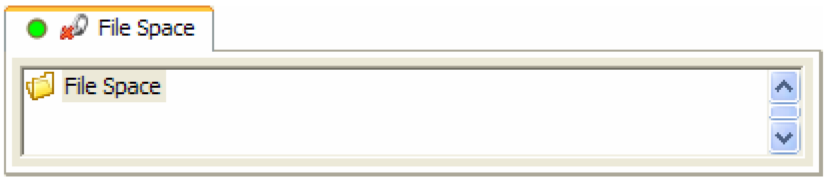
Type the name of the new root directory or click **Browse** to select a new directory using the standard file dialog box; then click **OK**.

2.8 File Space

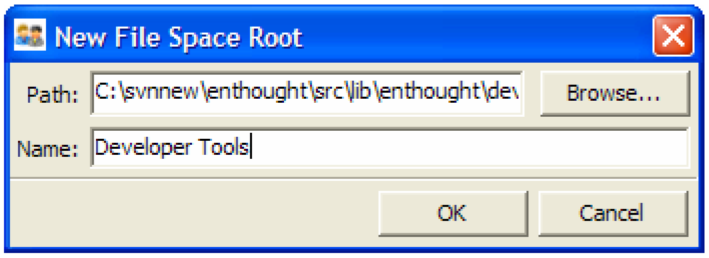
The File Space tool displays Python source files contained in possibly disjoint parts of the file system. Here is a screen shot of the File Space tool in use:



The first time the File Space tool is opened, the view is empty:



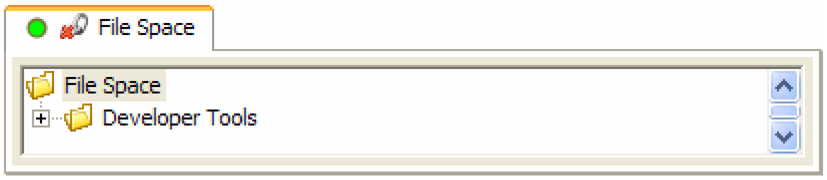
To add something to it, right-click the **File Space** root node in the tree view and selecting **New** > **File Space Root** on the shortcut menu, which displays the following dialog box:



Path: C:\svnnew\enthought\src\lib\enthought\dev

Name: Developer Tools

Type the path to the Python source files you want to add, or click **Browse** and select the path using the standard file system dialog box. Enter a **Name** to use as an alias for the path in the tree view and click **OK**. A new item with the **Name** you specified appears in the tree view as a child of the root **File Space** node:



You can continue to add as many entries as you like. The entries are persisted across sessions, so you can build up a list of favorite source code repositories.

You can use the File Space tool as a file source for other tools by doing any of the following:

- Drag an item in the File Space tree view and drop it on another tool.
- Select an item in the tree view, then drag the File Space tool's drag icon (●) and drop it on another tool.
- Connect the File Space tool to another tool by dragging or clicking the tool's connect icon ().

The File Space tool has four types of selection:

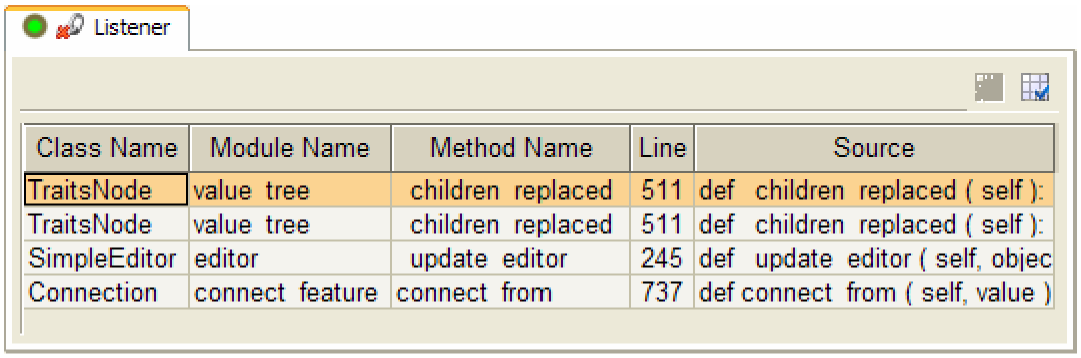
- The name of the most recently selected file.
- The name of the most recently selected directory.
- The name of the most recently selected path (either file or directory).
- A **FilePosition** for the most recently selected file.

Connecting the File Space tool to another tool connects one of these selection types to the other tool. Each time you select a new item, it is automatically sent to the connected tool.

2.9 Listener

The Listener tool displays all listeners currently associated with a specified object trait.

Here is a screen shot of the Listener tool in use:



Class Name	Module Name	Method Name	Line	Source
TraitsNode	value_tree	children_replaced	511	def _children_replaced (self):
TraitsNode	value_tree	children_replaced	511	def _children_replaced (self):
SimpleEditor	editor	update_editor	245	def _update_editor (self, objec
Connection	connect_feature	connect_from	737	def connect_from (self, value)

A class derived from **HasTraits** can have any number of listeners attached to each defined object trait. The listeners can be:

- Static listeners defined by the class itself using the _*traitname*_**changed** method naming convention.
- Dynamic listeners attached by other objects using the **on_trait_change** method.
- Dynamic listeners added implicitly by **HasTraits** through mechanisms such as the _*traitname*_**changed_for**_*object* method naming convention.

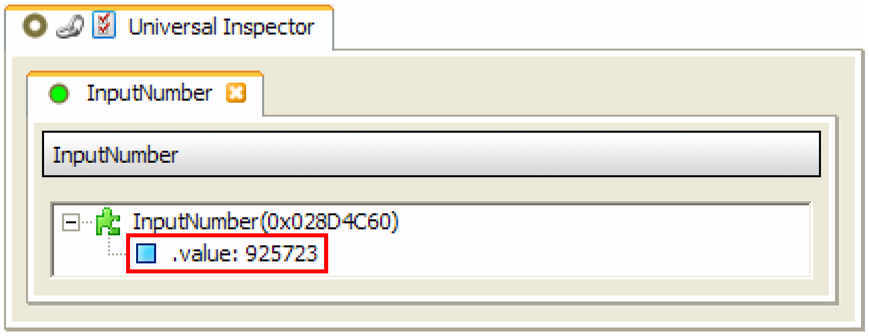
Using trait listeners is a very powerful and flexible technique for creating robust and scalable applications. However, it can sometimes lead to situations where hard to understand interactions occur between various objects because of the seemingly *invisible* listener connections between them.

For example, the screen shot above shows the trait listeners associated with the **value** trait of the trivial **InputNumber** class created as an example for one of the other sections of this document. The four listeners displayed include:

- Listeners created by the Universal Inspector tool to monitor and update the contents of the **InputNumber** object.
- Listeners created by the Traits UI **TextEditor** to synchronize the contents of the trait value and the text editor control.
- Listeners created by the **Connect** feature to connect the **value** trait of the **InputNumber** object to the **number** trait of a **ConvertNumber** object displayed in a separate Envisage view.

Thus, being able to see the connections between objects can be an important debugging or program understanding tool.

You specify the trait whose listeners you want to be displayed by dragging an object trait and dropping it on the Listener tool's drag and drop icon (●). Typical sources for this type of information are objects displayed in the Universal Inspector tool or by various views in the FBI debugger, as shown in the following figure:



This screen shot shows the trait (highlighted in red) that was dragged to the Listener tool to create the original screen shot.

The Listener tool shows a table of listeners connected to the specified object trait, one listener per row. Each table row can display the following columns:

- **Class Name**: The class name of the listener object.
- **Object ID**: The object ID of the listener object.
- **File Name**: The fully qualified name of the source file containing the listener object's listener method.
- **Module Name**: The name of the module containing the listener object's listener method.

- **Method Name**: The name of the listener object's listener method.
- **Line**: The line number within the source file where the object listener's listener method begins.
- **Source**: The contents of the first line of the object listener's listener method definition (usually a **def** statement).

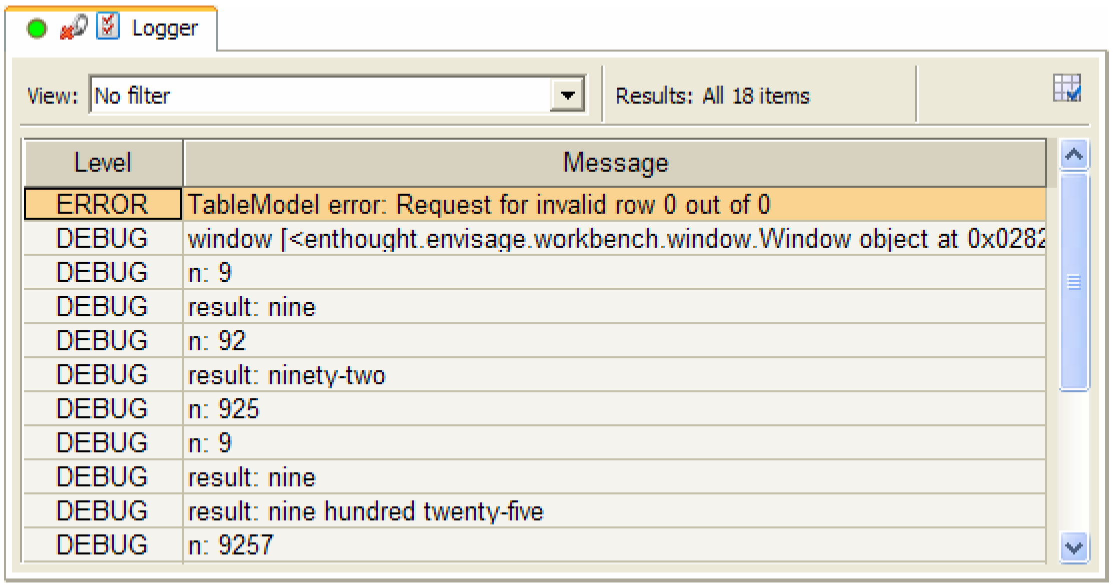
You can reorganize, add, and delete columns by clicking the Listener tool's user preference icon () in the table toolbar. You can also sort the rows by clicking the header of the column you want to sort. Reverse the order of the rows by clicking the same column header again. You can restore the original, unsorted order by clicking the unsort icon () in the table toolbar.

Selecting a row in the table also selects the corresponding listener object and listener method. You can access the currently selected listener object by dragging the Listener tool's drag and drop icon (●) and dropping it on another tool, such as the Universal Inspector. You can access the currently selected listener method, represented as a **FilePosition** object, by dragging the Listener tool's drag and drop icon to another tool, or by connecting the Listener tool to another tool by dragging or clicking its connect icon (). For example, connecting the Listener tool to the FBI Viewer can be very handy for viewing and setting breakpoints in various listener methods by selecting the corresponding rows in the Listener tool.

The Listener tool supports dragging both the currently selected listener object and listener method **FilePosition** object at the same time. The result of dropping the selection on another tool depends upon the tool. For example, the Universal Inspector understands both normal objects and **FilePosition** objects, and so adds two new sub-tabs: one that displays the contents of the listener object, and another that displays the source code of the listener method. Another tool might recognize only one or the other of the two objects, and will handle the object it understands. If a tool cannot process either object, the invalid drop target cursor is displayed.

2.10 Logger

The Logger tool displays messages logged using the standard Python library logging module. The following screen shot shows an example of the Logger tool in use:



Level	Message
ERROR	TableModel error: Request for invalid row 0 out of 0
DEBUG	window [<enthought.envisage.workbench.window.Window object at 0x0282
DEBUG	n: 9
DEBUG	result: nine
DEBUG	n: 92
DEBUG	result: ninety-two
DEBUG	n: 925
DEBUG	n: 9
DEBUG	result: nine
DEBUG	result: nine hundred twenty-five
DEBUG	n: 9257

Calls to the Python logging facility are typically inserted into source code by developers in order to capture significant program events. They can also be inserted into code dynamically using the FBI Viewer tool. Logging messages are divided into five categories, or levels:

- DEBUG
- INFO
- WARNING
- ERROR
- CRITICAL

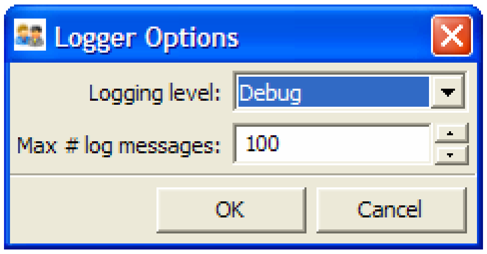
The Logger tool uses a tabular format to display the logging messages, with each row representing a single message. The table can display the following columns:

- **Level**: The level of the message (e.g., ERROR).
- **Message**: The message logged.

- **Level #**: The level number of the message (DEBUG = 10, INFO = 20, WARNING = 30, ERROR = 40, CRITICAL = 50).
- **File Name**: The file name of the module that logged the message.
- **Path Name**: The directory of the module that logged the message.
- **Module**: The name of the module that logged the message.
- **Line #**: The line number of the call to the logging module.
- **Logger Name**: The name of the logger object.
- **Process ID**: The process ID of the program logging the message.
- **Thread ID**: The ID of the Python thread that the logging call was made from.
- **Time Created**: The time at which the logging call was made in the format `YYYY-MM-DD HH:MM:SS,mmm`.
- **Milliseconds**: Millisecond portion of the time at which the logging call was made.
- **Raw Time Created**: The raw time at which the logging call was made, as returned by **time.time()**.

By default, only the **Level** and **Message** columns are displayed. However, you can reorganize, add and delete columns by clicking the user preference icon () in the table toolbar.

The various logging levels have a defined order of importance, with DEBUG being the lowest, and CRITICAL being the highest. By default, the Logger shows the 100 most recent log messages of any level of importance. You can change these settings by clicking the Logger's option icon (), which displays the following dialog box:



Logging level: Debug
Max # log messages: 100

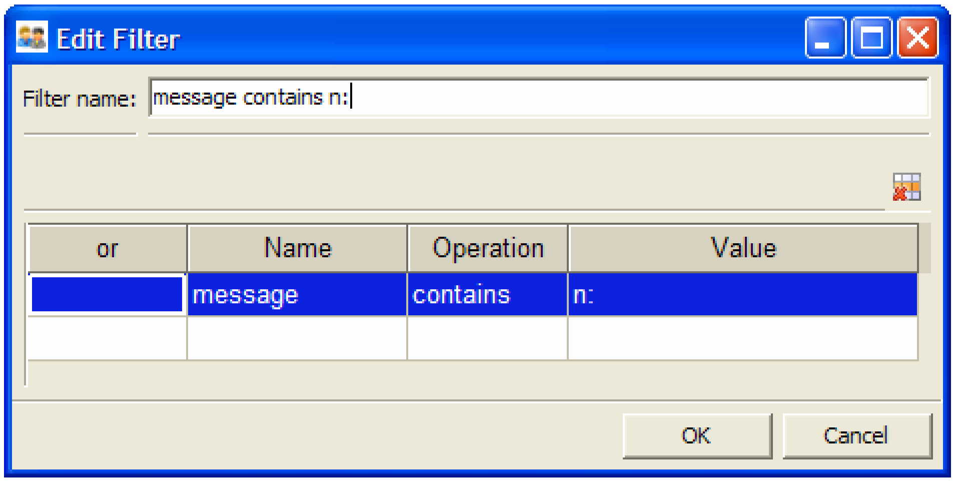
You can change the **Logging level** to any of the five standard levels. Only messages with the same (or higher) importance as the **Logging level** are displayed.

You can also change **Max # log messages** to reflect the maximum number of logging messages that can be displayed at any one time.

If that number is exceeded, the oldest messages are removed from the list. After message have been discarded, they cannot be recovered by increasing **Max # log messages**.

You can also *filter* the messages displayed in the Logger by creating a custom table filter. To create a filter, select **Customize** in the **View** list above the table. Any filter you create persists until you explicitly delete it (even if it is not currently applied).

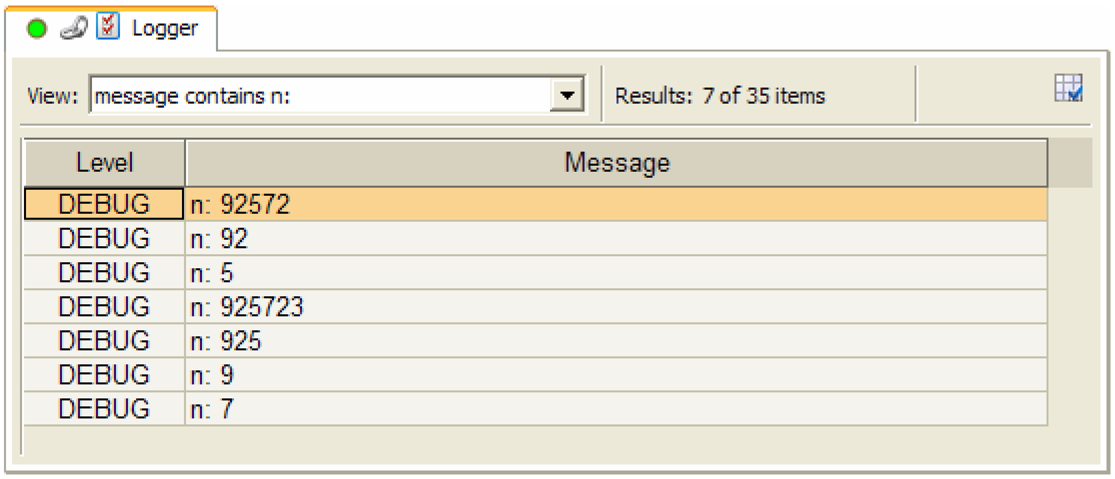
For example, here is a screen shot that shows the definition of a *rule-based* filter:



Filter name: message contains n:

or	Name	Operation	Value
	message	contains	n:

Here is a screen shot that shows the results of applying this filter in the Logger tool:



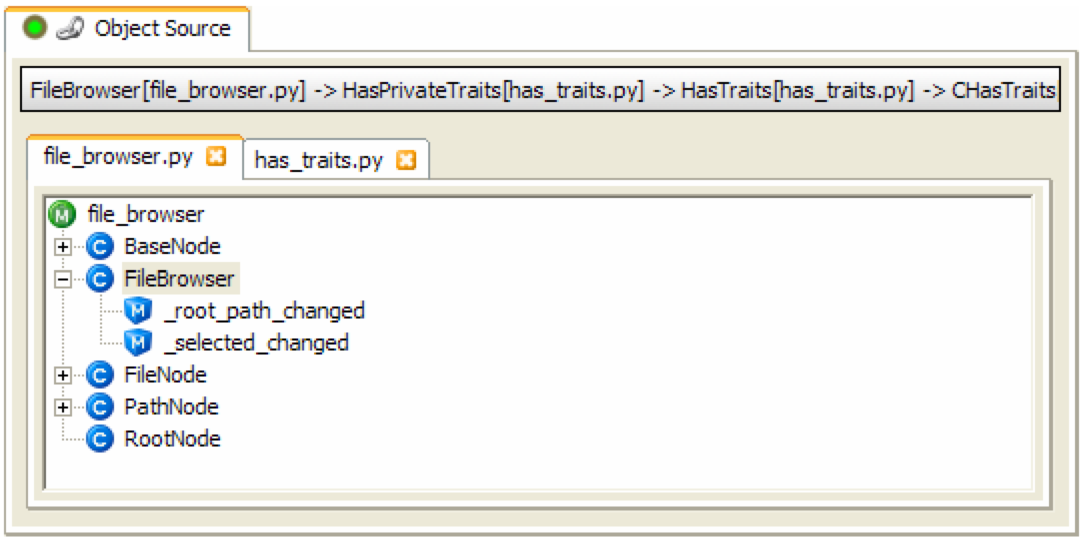
View: message contains n: Results: 7 of 35 items

Level	Message
DEBUG	n: 92572
DEBUG	n: 92
DEBUG	n: 5
DEBUG	n: 925723
DEBUG	n: 925
DEBUG	n: 9
DEBUG	n: 7

To remove the effects of a filter, select **No filter** in the **View** list. Unlike changing the **Max # log messages** option, filters have no permanent effect on the contents of the Logger table.

2.11 Object Source

The Object Source tool displays the hierarchy of modules, classes, and methods for each class that a specified Python object is derived from. Here is a screen shot of the Object Source tool in use:


FileBrowser[file_browser.py] -> HasPrivateTraits[has_traits.py] -> HasTraits[has_traits.py] -> CHasTraits

The message bar at the top of the tool shows the class (**FileBrowser**) and module (`[file_browser.py]`), for each class an object is derived from. Below that is a notebook with sub-tabs for each unique source file in the object's class derivation hierarchy.

Each notebook tab displays a tree view showing all of the classes and methods within the corresponding Python module. Selecting any node in the tree also selects a **FilePosition** object whose text range depends upon the type of item selected:

- **Module**: Entire source file
- **Class**: Entire class definition
- **Method**: Entire method definition

You can specify the object whose source you want to view by doing one of the following:

- Drag an object from another tool or view and drop it on the Object Source tools drop icon (●).
- Connect the Object Source tool to another tool that supplies objects by dragging or clicking the Object Source tool's connect icon ().

Like many of the tools in the **enthought.developer.tools** package, the Object Source tool is most useful when used in combination with other tools. For example, you are trying to find a bug on one of your Envisage views, so you might do the following:

1. Open the Application Monitor, which shows a tree of all the views in the current application.

2. Drag the Python object associated with the view you want to debug and drop it on the Object Source tool's drop icon, to display the object's class and method hierarchy.

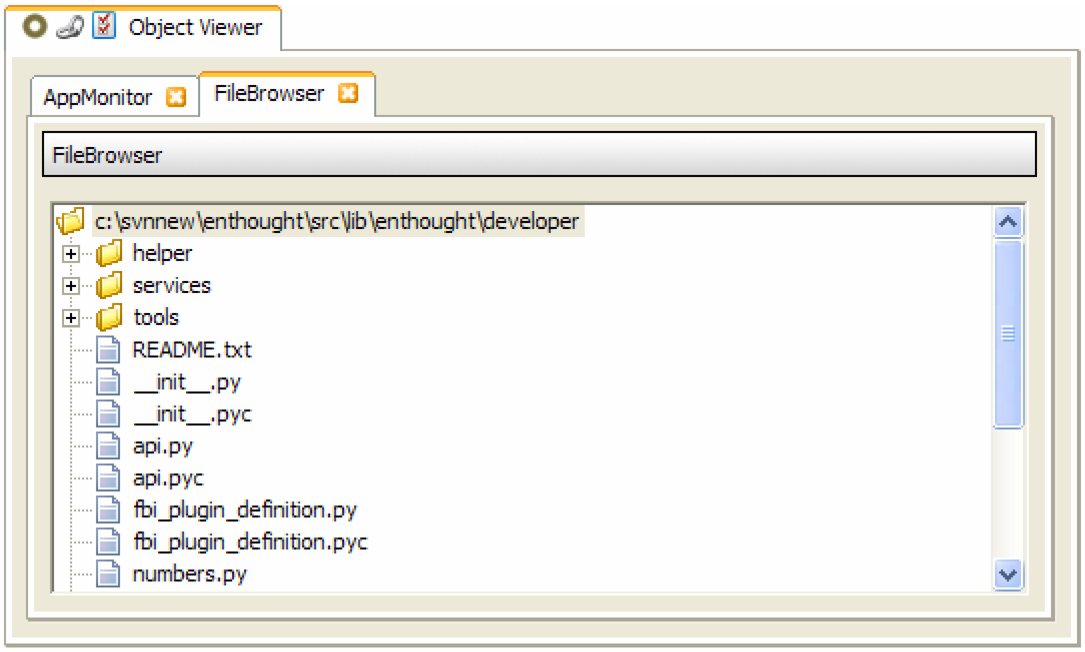
3. Drag the Object Source's connect icon and drop it on the FBI Viewer's connect icon.

4. Select a method name in the Object Source view and set some breakpoints in the method in the FBI Viewer.

Another related scenario is that, having finally debugged the Envisage view, you are now trying to determine why certain actions take so long to perform. So you again drag the view object from the Application Monitor to the Object Source tool, but this time connect the Object Source tool to the Profiler tool in order to profile the action code to find where the bottleneck is.

2.12 Object Viewer

The Object Viewer tool allows you to display the default Traits UI view for a specified object derived from **HasTraits**. Here is a screen shot of the Object Viewer tool in use:



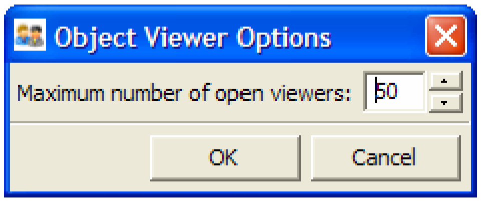
```
c:\svnnew\enthought\src\lib\enthought\developer
  helper
  services
  tools
  README.txt
  __init__.py
  __init__.pyc
  api.py
  api.pyc
  fbi_plugin_definition.py
  fbi_plugin_definition.pyc
  numbers.py
```

To add objects to the Object Viewer, do one of the following:

- Drag and drop an object with traits onto the Object Viewer's drop icon (●).
- Connect a tool that supplies objects with traits to the Object Viewer by dragging or clicking the viewer's connect icon ( ).

When you add an object to the Object Viewer, the tool creates a sub-tab containing the default Traits UI view for the object. You can remove any sub-tab by clicking its close icon ( ).

By default, the Object Viewer tool displays up to 50 tabs before it begins to automatically close the oldest tabs. To change this number, click the Object Viewer's option icon ( ), which displays the following dialog box:

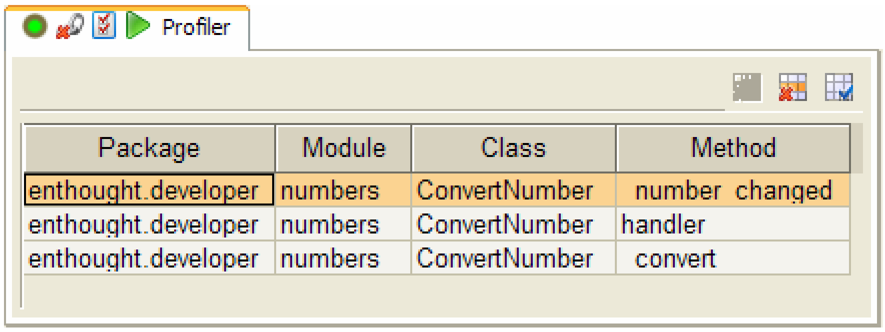


Maximum number of open viewers: 50

Change **Maximum number of open viewers** to the new value and click **OK**. If the new value is less than the current number of tabs, the tool closes the appropriate number of the oldest tabs automatically.

2.13 Profiler

The Profiler tool allows you to *profile* (gather execution time statistics) for one or more classes or methods in an Envisage application using the standard Python HotShot profiler. Here is a screen shot of the Profiler tool waiting to begin profiling:



Package	Module	Class	Method
enthought.developer	numbers	ConvertNumber	number_changed
enthought.developer	numbers	ConvertNumber	handler
enthought.developer	numbers	ConvertNumber	convert

The Profiler contains a table of all methods that are currently set to be profiled.

Each row represents a single method and consists of the following columns:

- **Package**: The name of the package that contains the method.
- **Module**: The name of the module that contains the method.
- **Class**: The name of the class that contains the method.
- **Method**: The name of the method to profile.

To add methods to the Profiler do one of the following:

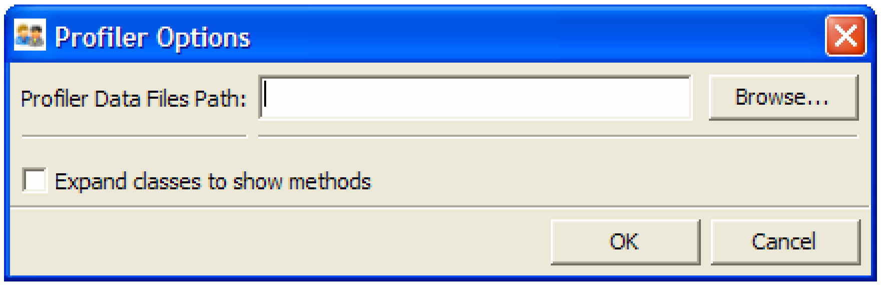
- Drag a selection from a tool that supplies methods (or classes), such as the Class Browser or Object Source tool, and drop it on the Profiler's drag and drop icon (●).
- Connect the Profiler to a source of methods (or classes) by dragging or clicking the Profiler tool's connect icon (🔌).

You can add classes as well as methods to the Profiler. Adding a class is the same as adding every method defined in the class, excluding any methods that the class inherits from its base classes.

Once one or more methods have been added to the Profiler, the start profiling icon (▶) appears on the Profiler's tab to indicate that it is ready to start profiling. Click the icon whenever you are ready profile your code. After clicking the icon, profiling begins and the icon changes to the stop profiling icon (🟥). Click this icon to stop profiling and analyze the results. After you click the stop profiling icon, the icon reverts to the start profiling icon to indicate that the Profiler is ready to start profiling again.

Each time the Profiler gathers profiling statistics, it writes the data to a file with a name of the form: `profiler_`*nnn*`.prof`, where *nnn* is an integer that is one larger than the last Profiler file created.

By default, the Profiler creates these files in the current directory, but you can specify a different path by clicking the Profiler's options icon (☑), which displays the following dialog box:



Type the name of the new path in **Path for Profiler data files**, or click **Browse** and use the system file dialog box to select the path. The **Expand classes to show methods** option is not currently implemented, so changing its value has no effect.

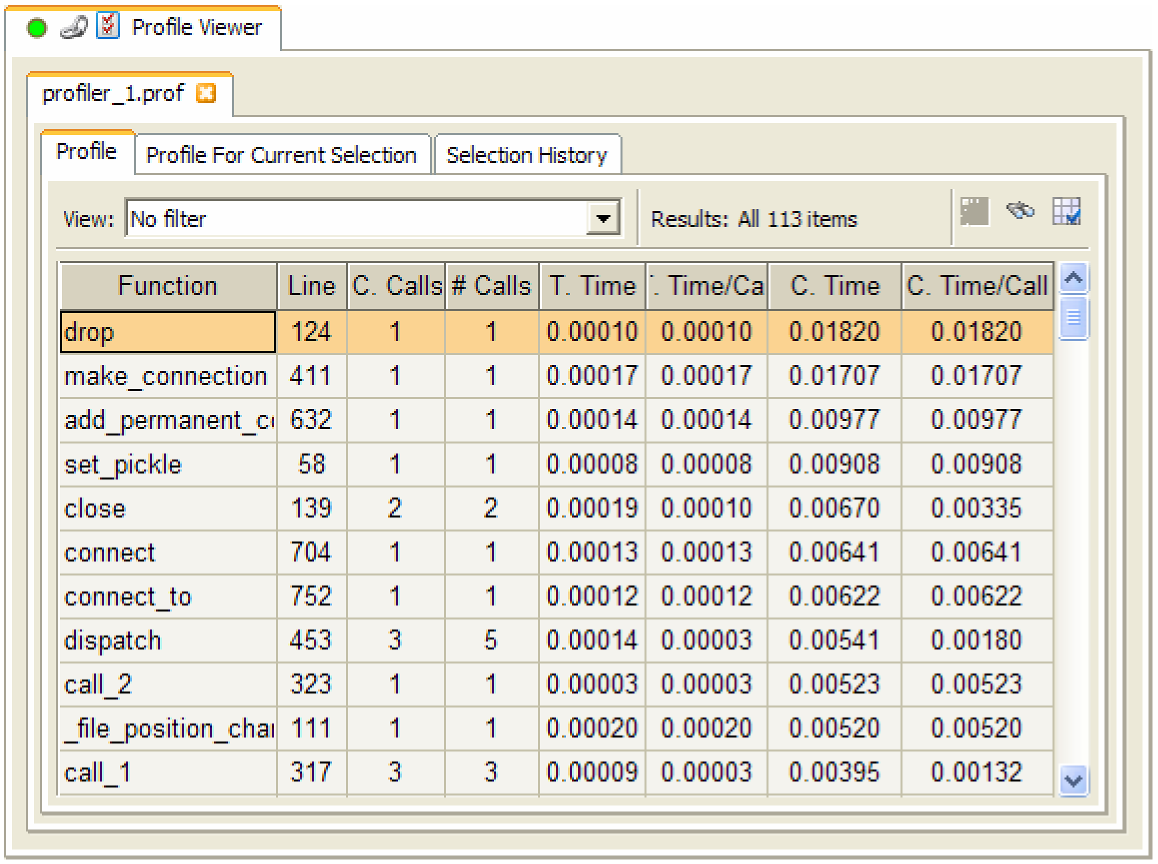
The Profiler does not display the analysis of the execution statistics it gathers. It simply saves the results to a file. You can make the name of the file available to other tools by doing one of the following:

- Drag the Profiler's drag and drop icon (●) to another tool that accepts file names, such as the Profile Viewer tool.
- Connect the Profiler to another tool that accepts file names, such as the Profile Viewer, by dragging or clicking the Profiler's connect icon ().

A typical usage scenario obviously is to drag or connect the output of the Profiler tool to the Profile Viewer tool, which is explicitly designed to analyze and display the contents of HotShot profile files. However, you can use whatever tool you prefer to process the Profiler results.

2.14 Profile Viewer

The Profile Viewer tool analyzes and displays the results of a Python Hotshot profiling session, such as those produced using the Profiler tool. Here is a screen shot of the Profile Viewer in use:



View: No filter

Results: All 113 items

Function	Line	C. Calls	# Calls	T. Time	. Time/Ca	C. Time	C. Time/Call
drop	124	1	1	0.00010	0.00010	0.01820	0.01820
make_connection	411	1	1	0.00017	0.00017	0.01707	0.01707
add_permanent_c	632	1	1	0.00014	0.00014	0.00977	0.00977
set_pickle	58	1	1	0.00008	0.00008	0.00908	0.00908
close	139	2	2	0.00019	0.00010	0.00670	0.00335
connect	704	1	1	0.00013	0.00013	0.00641	0.00641
connect_to	752	1	1	0.00012	0.00012	0.00622	0.00622
dispatch	453	3	5	0.00014	0.00003	0.00541	0.00180
call_2	323	1	1	0.00003	0.00003	0.00523	0.00523
_file_position_cha	111	1	1	0.00020	0.00020	0.00520	0.00520
call_1	317	3	3	0.00009	0.00003	0.00395	0.00132

The Profile Viewer is organized as a series of notebook tabs, with one tab for each profiling session file being displayed. Each tab in turn contains three notebook sub-tabs:

- **Profile**: Displays all profiling statistics.
- **Profile for Current Selection**: Displays the profiling statistics only for those methods and functions called from the selected item in the **Profile** tab.
- **Selection History**: Displays all previously selected **Profile** tab items.

Each of these tabs contains a table showing methods and functions called during the profiling session. Each row represents a single method or function and contains the following columns:

- **File Name**: The fully qualified file name of the module containing the profiled function.
- **Path**: The full path name of the module containing the profiled function.
- **Module**: The name of the module containing the profiled function.
- **Function**: The name of the profiled function.
- **Line**: The number of the line within the module where the definition of the profiled function begins.
- **C. Calls**: Cumulative number of calls to the profiled function, including recursive calls.
- # **Calls**: Number of calls made to the profiled function.
- **T. Time**: Total time spent in the profiled function, excluding time spent calling out to other functions.
- **T. Time/Call**: Average total time spent in the profiled function.
- **C. Time**: Cumulative time spent in the profiled function, including time spent calling out to other functions.
- **C. Time/Call**: Average cumulative time spent in the profiled function.

To re-organize, add and delete columns by clicking the user preference icon () in the table toolbar.

You can also sort the contents of the table by clicking a column header. Reverse the sort order by clicking the same column header again. You can restore the original table order by clicking the unsort icon () in the table toolbar.

You can filter the contents of the table in any number of ways by selecting a filter in the **View** list. You can also create new filters by selecting **Customize** in the **View** list. Any filters you create are automatically persisted until you explicitly delete them.

In a similar fashion, you can search the table for particular entries by clicking the search icon () in the table toolbar and entering an expression which evaluates to **True** when an item satisfies your search criteria. Using the buttons in the search dialog box, you can move forward and backward from one matching entry to the next or select all matching entries at once.

When entering a Python search expression, you must specify the correct attribute name for each column.

Table 2 Profile Viewer column attributes

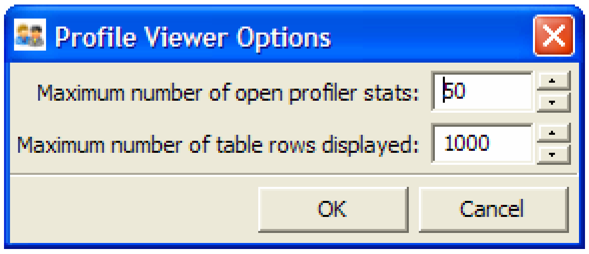
Column	*Attribute*
File Name	file_name
Path	path_name
Module	module_name
Function	function
Line	line
C. Calls	cumulative_calls
# **Calls**	n_calls
T. Time	total_time
T. Time/Call	total_time_per_call
C. Time	cumulative_time
C. Time/Call	cumulative_time_per_call

For example, the expression `n_calls >= 1000` matches all functions called at least 1000 times.

Each time you select a function in the **Profile** tab, the **Profile For Current Selection** tab is updated to show only functions called from the selected function, including itself. This behavior can be useful when you are trying to "drill down" into a particular section of the overall profile data.

Similarly, the first time you select an entry in the **Profile** tab, the entry is also added to the bottom of the **Selection History** table, creating a list of entries you selected. Selecting any entry in the **Selection History** tab causes the corresponding entry in the **Profile** tab to be reselected as well, which in turns causes the **Profile For Current Selection** tab to be updated. This behavior can be useful when you are analyzing different parts of the data and need to switch back and forth between the same subsets of the data.

By default, the Profile Viewer tool displays up to 50 different profiling session results, with up to 1000 entries per table. You can change either of these limits by clicking the tool's options icon (), which displays the following dialog box:


Profile Viewer Options

Maximum number of open profiler stats: 50

Maximum number of table rows displayed: 1000

Change either of the options to their new values and then click **OK**. Allowing a very large number of table rows can negatively affect the speed at which the display refreshes.

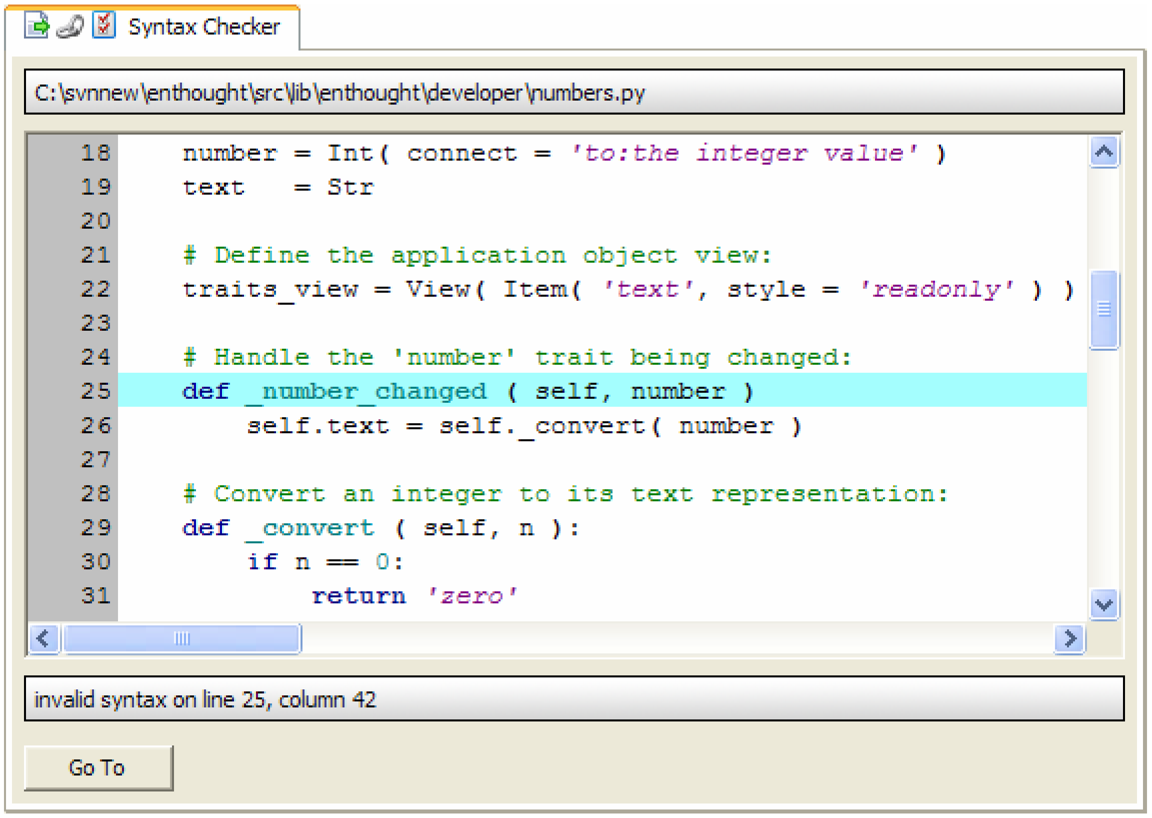
You can also delete the view of any profiling session by clicking the session tab's close icon ().

You can supply input to the Profile Viewer tool by doing one of the following:

- Drag a file containing HotShot profiling data and drop it on the Profile Viewer's drop icon (●).
- Connect the Profile Viewer to a source of HotShot profile data, such as the Profile tool, by dragging or clicking the Profile Viewer's connect icon ().

2.15 Syntax Checker

The Syntax Checker tool allows you to check Python source files for syntax errors. Here is a screen shot that shows the Syntax Checker in use:



C:\svnnew\enthought\src\lib\enthought\developer\numbers.py

```
    number = Int( connect = 'to:the integer value' )
    text   = Str

    # Define the application object view:
    traits_view = View( Item( 'text', style = 'readonly' ) )

    # Handle the 'number' trait being changed:
    def _number_changed ( self, number )
        self.text = self._convert( number )

    # Convert an integer to its text representation:
    def _convert ( self, n ):
        if n == 0:
            return 'zero'
```

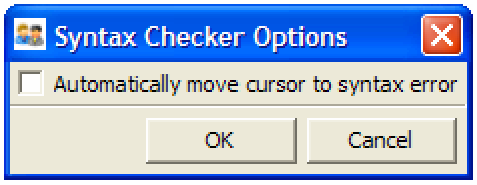
invalid syntax on line 25, column 42

The message bar at the bottom of the view shows where first syntax error in the module, if any, was found. To position the cursor to the point of error, clicking **Go To**. You can then type changes to correct the error. The Syntax Checker automatically re-parses the program and reports the next error it finds, if any. After you have corrected all syntax errors, the message bar displays "Syntactically correct":

Syntactically correct

If you have changed the source file, the save icon (💾) appears on the Syntax Checker tool's tab. Click the icon to save the changes to the source file; when the file is saved, the icon disappears.

By default, the Syntax Checker does not automatically move the cursor to the current syntax error. Instead, it waits for you to click **Go To**. However, you can change this behavior by clicking the options icon (☑) and selecting **Automatically move cursor to syntax error** in the options dialog box that is displayed:



☐ Automatically move cursor to syntax error

The Syntax Checker also defaults to automatically reloading the current source file if it is changed externally to the tool. If you do not want this behavior, clear **Automatically reload externally changed files** in the options dialog box.

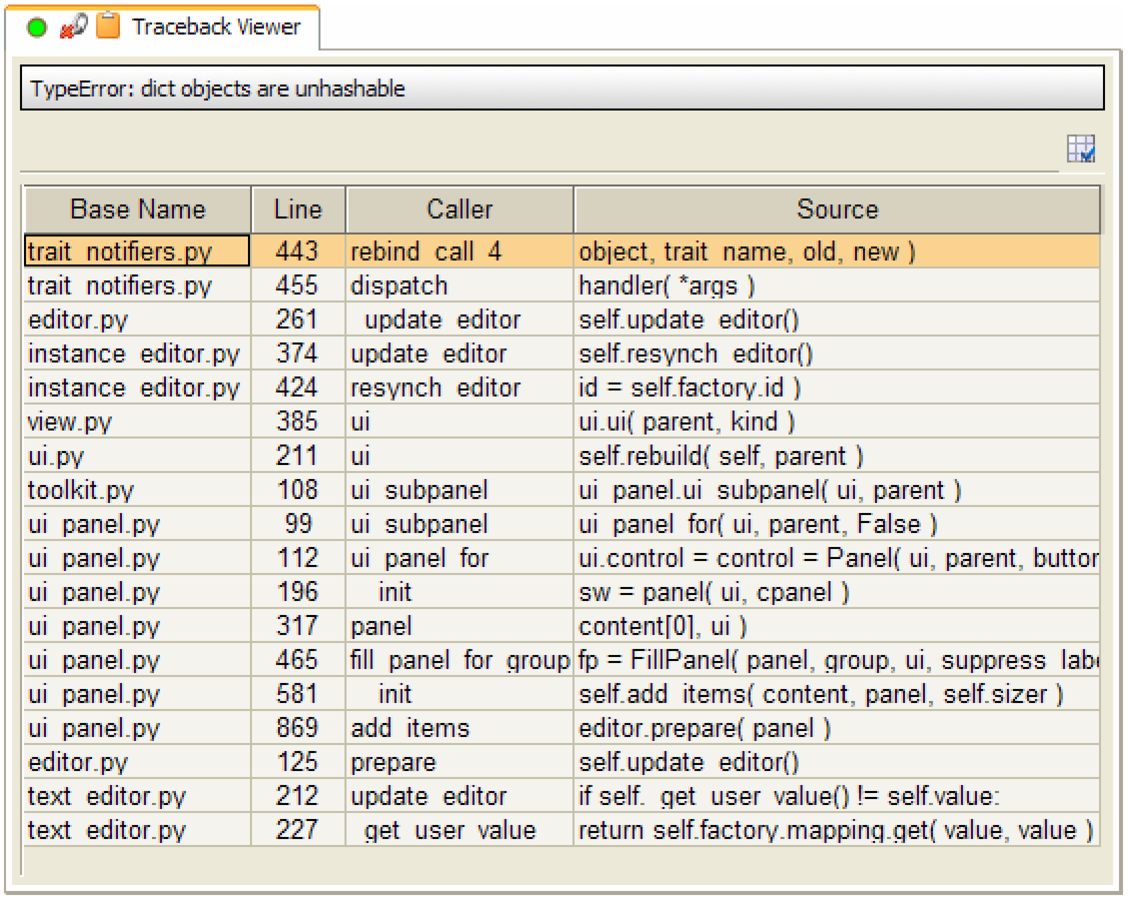
To select the Python source file to check, do one of the following:

- Drag a file from any tool or program that supplies file names and drop it on the drop file icon (). Examples of such file name sources are the Windows Explorer program and the File Browser tool.
- Connect another tool that supplies file names, such as the File Browser, to the Syntax Checker by dragging or clicking the connect icon ().

You can also drag the current source file being checked to another tool by dragging the drop file icon.

2.16 Traceback Viewer

The Traceback Viewer tool displays Python exception tracebacks in a formatted table. Here is a screen shot that shows the Traceback Viewer in use:



Traceback Viewer

TypeError: dict objects are unhashable

Base Name	Line	Caller	Source
trait notifiers.py	443	rebind call 4	object, trait name, old, new)
trait notifiers.py	455	dispatch	handler(*args)
editor.py	261	update editor	self.update editor()
instance editor.py	374	update editor	self.resynch editor()
instance editor.py	424	resynch editor	id = self.factory.id)
view.py	385	ui	ui.ui(parent, kind)
ui.py	211	ui	self.rebuild(self, parent)
toolkit.py	108	ui subpanel	ui panel.ui subpanel(ui, parent)
ui panel.py	99	ui subpanel	ui panel for(ui, parent, False)
ui panel.py	112	ui panel for	ui.control = control = Panel(ui, parent, buttor
ui panel.py	196	init	sw = panel(ui, cpanel)
ui panel.py	317	panel	content[0], ui)
ui panel.py	465	fill panel for group	fp = FillPanel(panel, group, ui, suppress lab
ui panel.py	581	init	self.add items(content, panel, self.sizer)
ui panel.py	869	add items	editor.prepare(panel)
editor.py	125	prepare	self.update editor()
text editor.py	212	update editor	if self. get user value() != self.value:
text editor.py	227	get user value	return self.factory.mapping.get(value, value)

The message bar at the top of the view displays the exception that occurred, and the table below it shows the traceback entries for the exception, with one row for each stack frame, with the most recently executed stack frame at the bottom of the table.

The table contains the following columns:

- **File Name**: Full path and file name of the module the stack frame was executing.
- **Path Name**: Full path name of the module the stack frame was executing.
- **Base Name**: Base name of the module the stack frame was executing.
- **Line**: Line number within the module the stack frame was executing.
- **Caller**: Method or function name the stack frame was executing.
- **Source**: Source code of the line the stack frame was executing.

By default, only the **Base Name**, **Line**, **Caller** and **Source** columns are displayed. To reorganize, add, and remove columns, click the user preference icon ( ) located to the top-right of the table.

To provide input (i.e., an exception traceback) to the Traceback Viewer, do one of the following:

- Drag and drop a traceback on the tool's drop icon (●).
- Connect a tool which supplies traceback information, such as the Logger tool, to the viewer's connect icon ( ).
- Copy a sequence of text lines containing traceback information and paste it into the viewer by clicking its paste icon ( ).

When you select a row in the viewer table, the tool also selects a **FilePosition** object that describes the source file location corresponding to the selected stack frame. The selected **FilePosition** object can be connected to any other tool that accepts **FilePosition** information, such as the FBI Viewer, by dragging or clicking the viewer's connect icon ( ).

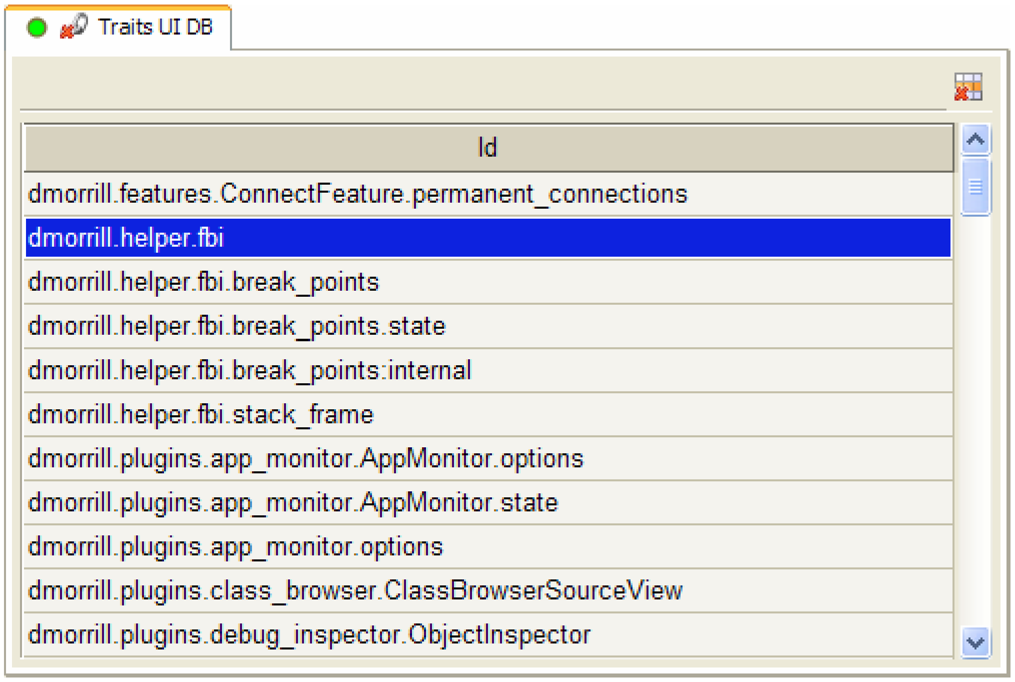
By itself, the ability to view a traceback in a tabular format is pleasant, but not compelling. The real usefulness of the Traceback Viewer becomes apparent when you combine it with other tools.

For example, a user sends you an e-mail message reporting a problem and includes the traceback they encountered. You copy the traceback text from the e-mail and click on the Traceback Viewer's paste icon to display the traceback. Next you drag the tool's connect icon and drop it on the FBI Viewer's connect icon to connect them together. Now you can select stack frames in the Traceback Viewer and view the corresponding source code in the FBI Viewer. You might even decide to try to reproduce the problem, and begin to set some breakpoints using the FBI Viewer.

Or maybe you are trying to track down problems in a plug-in you are developing, and you have configured the Logger tool to show logged exceptions. You connect the Logger tool to the Traceback Viewer so that you can select an exception in the Logger and view the corresponding traceback in the Traceback Viewer. Then, as in the previous scenario, you connect the Traceback Viewer to the FBI Viewer so that you can view the source code and set appropriate breakpoints.

2.17 Traits UI DB

The Traits UI DB tool allows you to browse, inspect, and delete items in the Traits UI data base. The Traits UI database contains persistent information, usually user preference data, about application views and windows created using the Traits UI API. Being able to browse, inspect, and sometimes delete this information can be very useful when you are trying to debug Traits-based user interface problems. Here is a screen shot of the Traits UI DB tool in use:



Id
dmorrill.features.ConnectFeature.permanent_connections
dmorrill.helper.fbi
dmorrill.helper.fbi.break_points
dmorrill.helper.fbi.break_points.state
dmorrill.helper.fbi.break_points:internal
dmorrill.helper.fbi.stack_frame
dmorrill.plugins.app_monitor.AppMonitor.options
dmorrill.plugins.app_monitor.AppMonitor.state
dmorrill.plugins.app_monitor.options
dmorrill.plugins.class_browser.ClassBrowserSourceView
dmorrill.plugins.debug_inspector.ObjectInspector

The tool lists the IDs of all current Traits UI database entries in alphabetical order. To delete an ID and its associated data from the database, select the ID and clicking the delete icon (▦) that appears on the top-right hand side of the list.

To inspect the data associated with a particular ID, first select the ID, and then do one of the following:

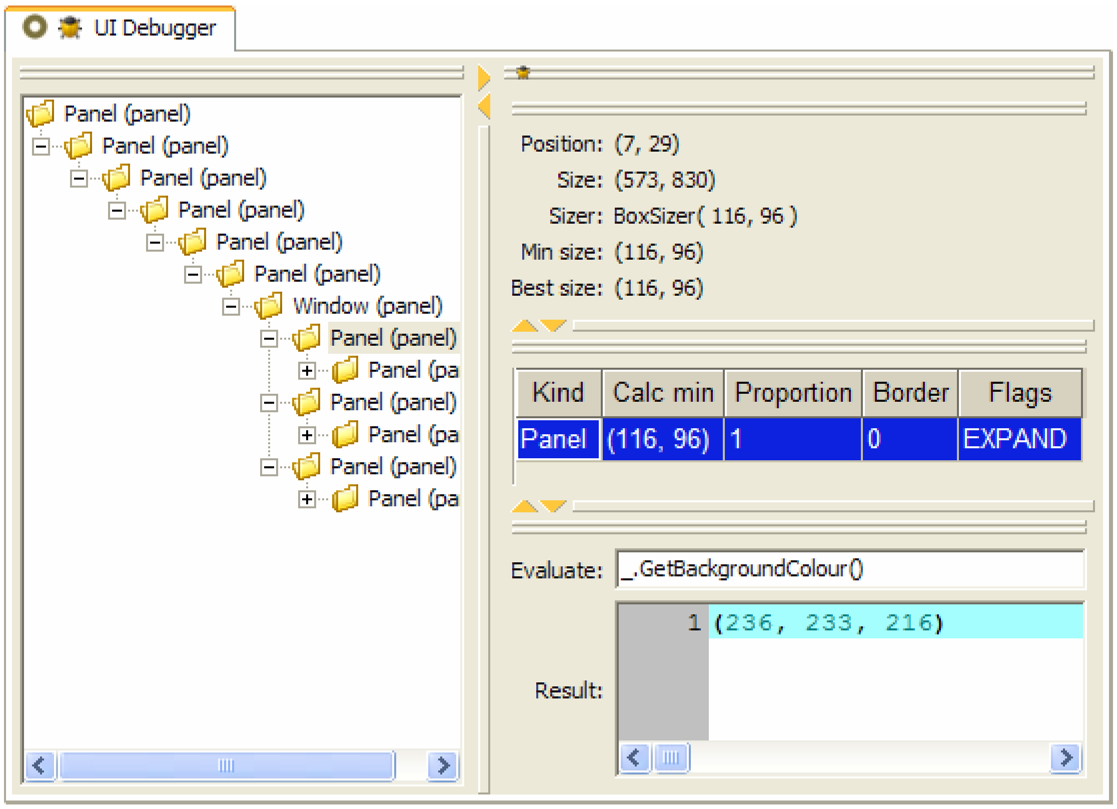
- Drag the drag icon (●) and drop it on another tool that can display Python objects, such as the Universal Inspector.

- Connect the Traits UI DB tool to another tool that can display Python objects by dragging or clicking the connect icon ( ). This action creates a permanent connection that allows you to select any database ID and immediately see its corresponding value in the other tool's view.

2.18 UI Debugger

The UI Debugger tool displays a tree view of a section of the application's wxPython window hierarchy starting at a specified top-most, parent window. It also allows you to display information about, query, or modify the contents of any window in the hierarchy.

Here is screen shot of the UI Debugger in use:



Position: (7, 29)
Size: (573, 830)
Sizer: BoxSizer(116, 96)
Min size: (116, 96)
Best size: (116, 96)

Kind	Calc min	Proportion	Border	Flags
Panel	(116, 96)	1	0	EXPAND

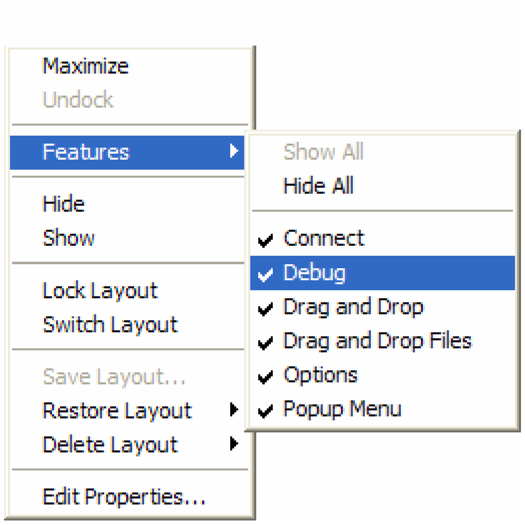
Evaluate: _.GetBackgroundColour()

Result:

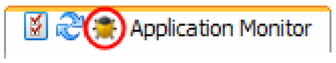
```
(236, 233, 216)
```

To specify the top-most parent window to display, do the following:

1. If the Debug feature is not already active, right click any notebook tab and click the **Debug** option of the **Features** sub-menu:

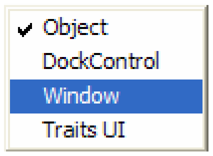


This action adds a **Debug** icon to each notebook tab that has an associated application object:

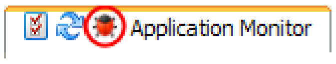


By default, the Debug icon represents the application object associated with the tab. But you can modify it to represent the wxPython window associated with the tab.

2. Right-click the Debug icon and click **Window** on the shortcut menu:

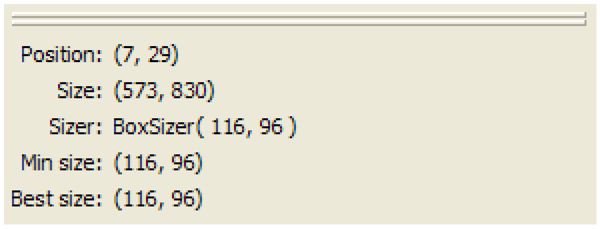


This modifies the **Debug** icon as shown below:



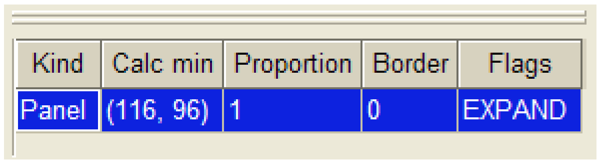
3. Drag a Debug window icon and drop it on the UI Debugger's drop icon (◉), icon to set the window hierarchy root.

After you have set the root window of the UI Debugger, the tool displays a tree view of all the children of the root window, as well as their children, and so on. If you select an item in the tree view, the tool updates the other three views with information about the selected window. For example, one view shows information about the position and size of the selected window:



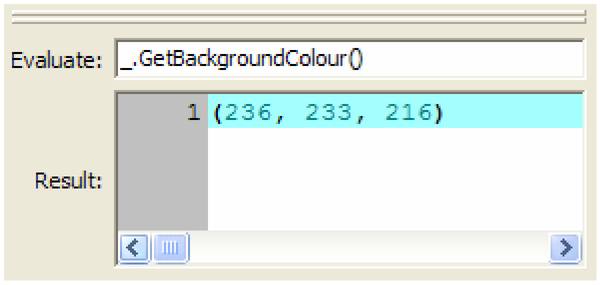
Position:	(7, 29)
Size:	(573, 830)
Sizer:	BoxSizer(116, 96)
Min size:	(116, 96)
Best size:	(116, 96)

Another view shows **wx.Sizer** information about the children of the selected window:



Kind	Calc min	Proportion	Border	Flags
Panel	(116, 96)	1	0	EXPAND

The remaining view allows you to interactively query and modify the selected window by entering Python code in the **Evaluate** field.



Evaluate: _.GetBackgroundColour()

Result:

```
(236, 233, 216)
```

The variable "_" is defined to be the selected window, while "__" is the **wx.Sizer** associated with the selected window. The results of any Python expression entered are shown below in the **Results** field. The expression is evaluated each time you type a character, so there is no need to press the Enter or Return key.

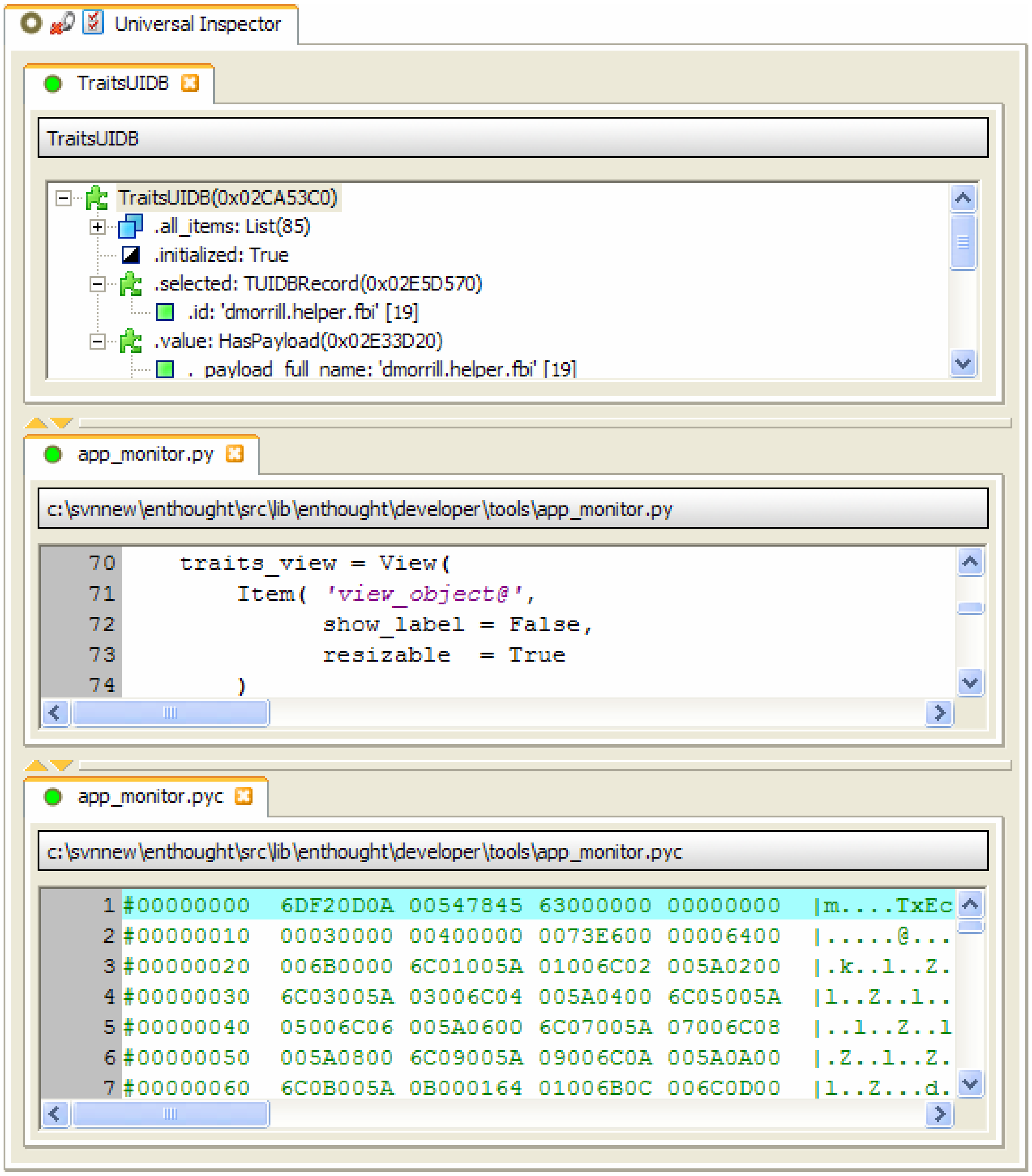
All of the sub-views are **DockWindow** components, and can be reorganized and resized as desired. Any new layout is automatically persisted for future uses of the tool.

2.19 Universal Inspector

The Universal Inspector tool displays a variety of data types, including:

- Python objects
- Python objects stored in pickle files
- Text files
- Binary files

Here is a screen shot that shows the Universal Inspector displaying the contents of an application, a Python source file, and the compiled form of the same Python source file.


Universal Inspector
TraitsUIDB
TraitsUIDB

```
TraitsUIDB(0x02CA53C0)
  .all_items: List(85)
  .initialized: True
  .selected: TUIDBRecord(0x02E5D570)
    .id: 'dmorrill.helper.fbi' [19]
  .value: HasPayload(0x02E33D20)
    .payload_full_name: 'dmorrill.helper.fbi' [19]
```

app_monitor.py
c:\svnnew\enthought\src\lib\enthought\developer\tools\app_monitor.py

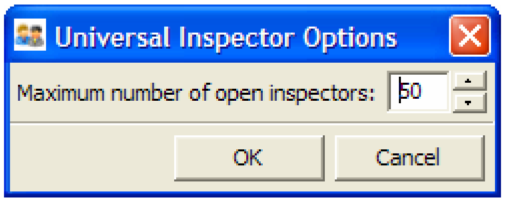
```
    traits_view = View(
        Item( 'view_object@',
              show_label = False,
              resizable  = True
        )
```

app_monitor.pyc
c:\svnnew\enthought\src\lib\enthought\developer\tools\app_monitor.pyc

```
#00000000  6DF20D0A 00547845 63000000 00000000  |m....TxEc
#00000010  00030000 00400000 0073E600 00006400  |.....@...
#00000020  006B0000 6C01005A 01006C02 005A0200  |.k..l..Z.
#00000030  6C03005A 03006C04 005A0400 6C05005A  |l..Z..l..
#00000040  05006C06 005A0600 6C07005A 07006C08  |..l..Z..l
#00000050  005A0800 6C09005A 09006C0A 005A0A00  |.Z..l..Z.
#00000060  6C0B005A 0B000164 01006B0C 006C0D00  |l..Z...d.
```

Each new item added to the Universal Inspector appears in a separate notebook tab. Like other **DockWindow** components, the tabs can be moved around and reorganized as desired, as the previous screen shot illustrates. Existing items can be deleted by clicking the item's delete icon (☒).

By default, the tool display up to 50 items before starting to delete the oldest items. However, you can change this value by clicking the tool's option icon (), which displays the following dialog box:



Maximum number of open inspectors: 50

Change **Maximum number of open inspectors** to the value you want, and then click **OK**. If the new value is smaller than the current number of open inspectors, the oldest items in excess of the new maximum are deleted.

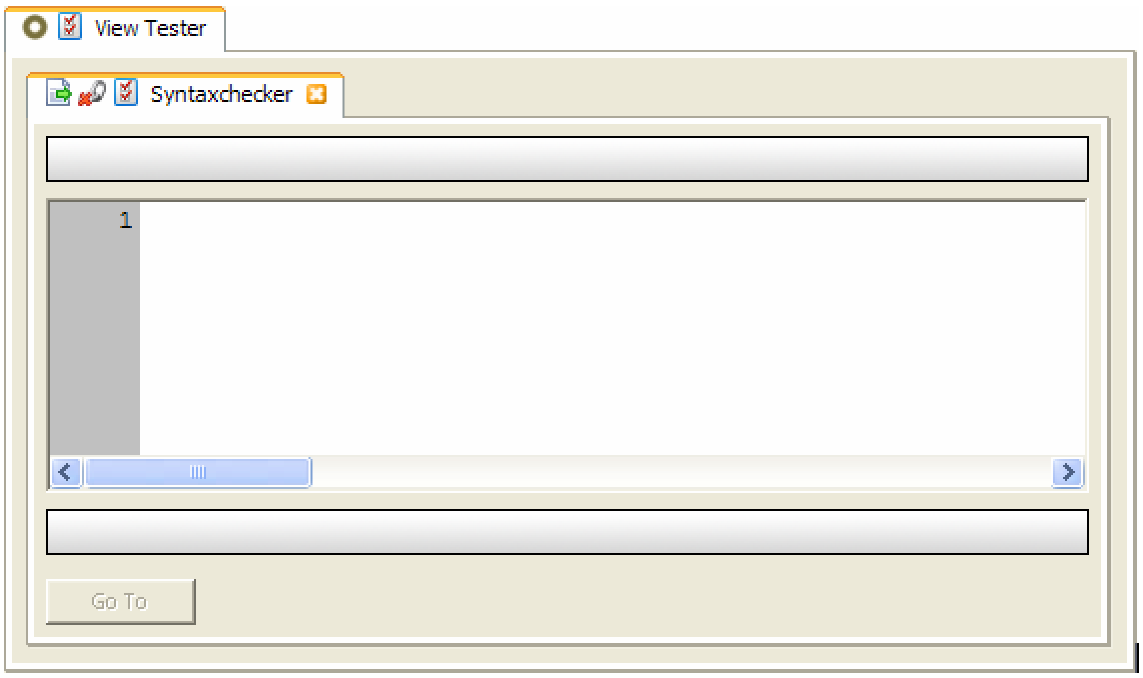
To add items to the Universal Inspector, do one of the following:

- Drag an item from another tool, such as the Traits UI DB tool, or another program, such as Windows Explorer, and drop it on the drop icon ().
- Connect the Universal Inspector to a tool that supplies items, such as the File Browser tool, by dragging or clicking the connect icon ().

Each item displayed in the Universal Inspector can also be dragged and dropped on other tools by dragging the item's drag icon (●).

2.20 View Tester

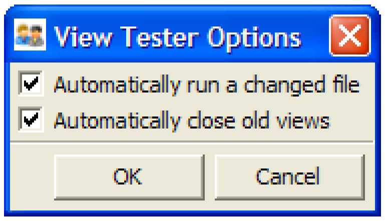
The View Tester tool allows you to quickly iterate on the design of a Traits UI-based Envisage view. Here is a screen shot of the View Tester being used to test the Syntax Checker too:



To use the View Tester tool, drag a Python source file and drop it on the tester's drop icon (●). The only requirement on the source file is that it must contain a module-level object derived from **HasTraits** called either `view` or `demo`. The View Tester tool loads the file, locates the object named `view` or `demo`, and then opens and displays the object's default trait view as a notebook tab within the tool.

The View Tester also monitors the dropped source file for changes, and when a change occurs, it automatically closes the previous view's tab, reloads the source file, and creates a new tab containing the view for the changed file. This behavior facilitates rapidly iterating on user interface and logic changes. You can makes changes in your text editor, and then immediately see the effect of the changes when you save the file.

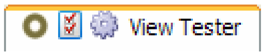
If you do not want the previous version of a view to be closed automatically after the source file changes, or you do not want the changed code to be automatically loaded by the View Tester, click the tester's option icon (☑), which displays the following dialog box:



[✓] Automatically run a changed file
[✓] Automatically close old views

Change the appropriate option, and then click **OK**. For example, you might want to clear **Automatically close old views** if you want to compare the *before* and *after* effects of a source file change. With the option cleared, you can see several versions of the view simultaneously. Of course, you can remove a view from the tool at any time by clicking the delete icon (☒) on its tab.

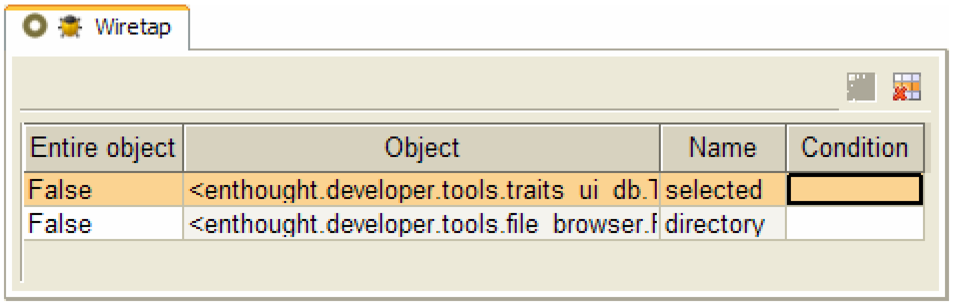
If you clear **Automatically run a changed file**, changes to the source file do not automatically cause a new view to be created. Instead, the load icon (⚙) appears on the View Tester's tab:



Click the load icon when you are ready to test the latest version of the source code. When you click the icon, the tool creates a new view tab based on the current source file, and also removes the icon from the tab until the source file is changed again. Working in this mode is useful when you make lots of source file changes and save the file frequently.

2.21 Wiretap

The Wiretap tool allows you to set and remove breakpoints based on the value of a trait being changed. Here is a screen shot of the Wiretap tool in use:



Entire object	Object	Name	Condition
False	<enthought.developer.tools.traits_ui_db.T	selected	
False	<enthought.developer.tools.file_browser.F	directory	

To create a wiretap breakpoint, drag a trait value from a tool such as the Universal Inspector and drop it on the Wiretap tool's drop icon (●). Each time the specified trait changes value, the FBI debugger opens, displaying the source code line that changed the trait value.

The Wiretap tool displays a table containing an entry for each currently defined wiretap breakpoint. The table has the following columns:

- **Entire Object**: If set to **True**, a wiretap breakpoint is triggered each time that any trait of the specified object changes value. Otherwise the breakpoint is triggered only when the specified trait changes value.
- **Object**: Contains a description of the object the wiretap breakpoint is set on.
- **Name**: The name of the object trait the wiretap breakpoint is set on.
- **Condition**: An optional Python expression that is evaluated each time the specified trait changes value. If the expression evaluates to **True**, the wiretap breakpoint is triggered; otherwise execution continues. If no expression is specified, the breakpoint is triggered every time the trait changes value.

To modify the **Entire Object** and **Condition** fields, select the field in the table and enter the new value.

To delete a wiretap breakpoint, select the breakpoint in the table and click the delete icon (▦) in the table toolbar.

To sort the items in the table, click any column header. Reverse the sort order of the data by clicking the same column header again. To un-sort the data, click the unsort icon (▦) in the table toolbar.

Unlike the breakpoints set by the Break Points tool, a wiretap breakpoint is not persisted across sessions because it depends upon monitoring an object which either will not exist or will have a different identity the next time the application is run.